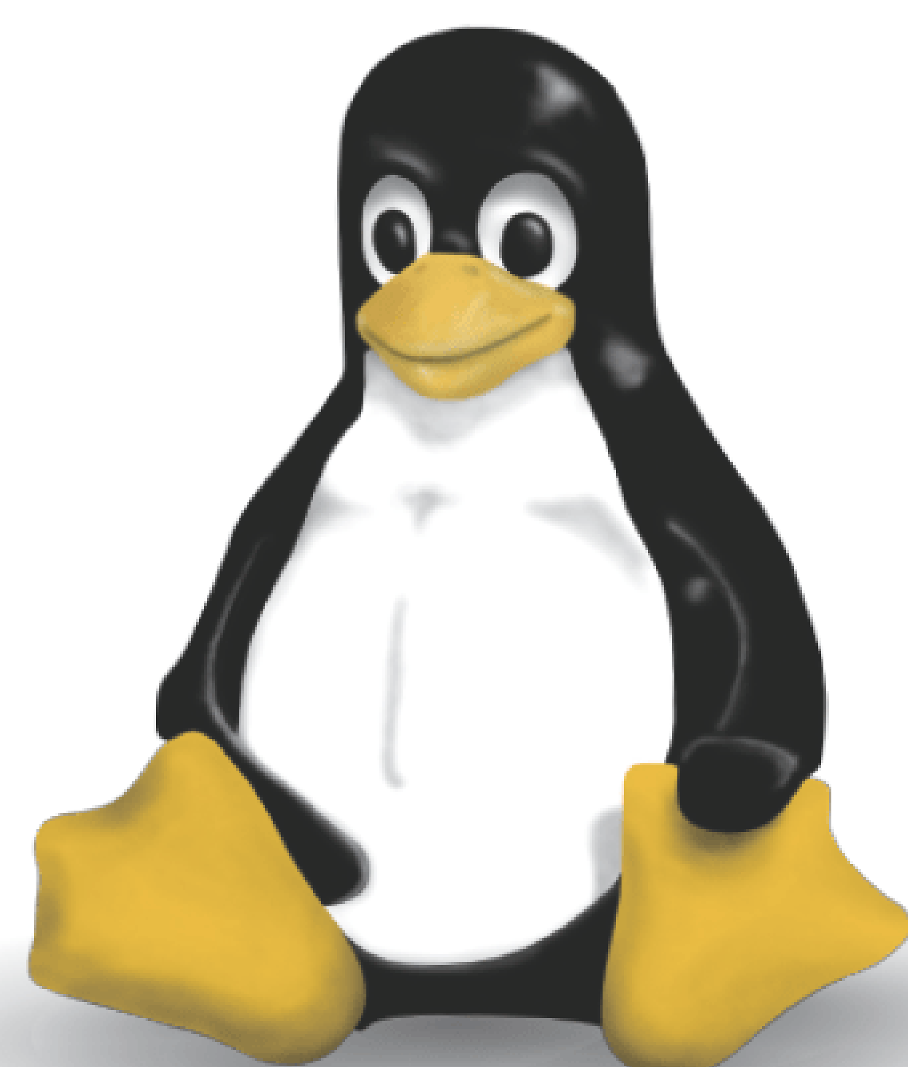
- 循序渐进：从最基本的Linux基础知识和分类开始，逐步介绍在Linux中编写C语言的方法、Linux的文件、流、进程、线程、网络编程、图形编程等知识，以及如何使用C语言进行相应的实现和操作。
- 实例丰富：本书对于介绍的相应知识都基于Linux的Ubuntu发行版给出了大量的实例，以及两个综合案例。
- 辅助视频：本书录制了长达10多个小时的视频，由浅入深地介绍了Linux的基础使用方法以及相应的C语言函数调用方法。



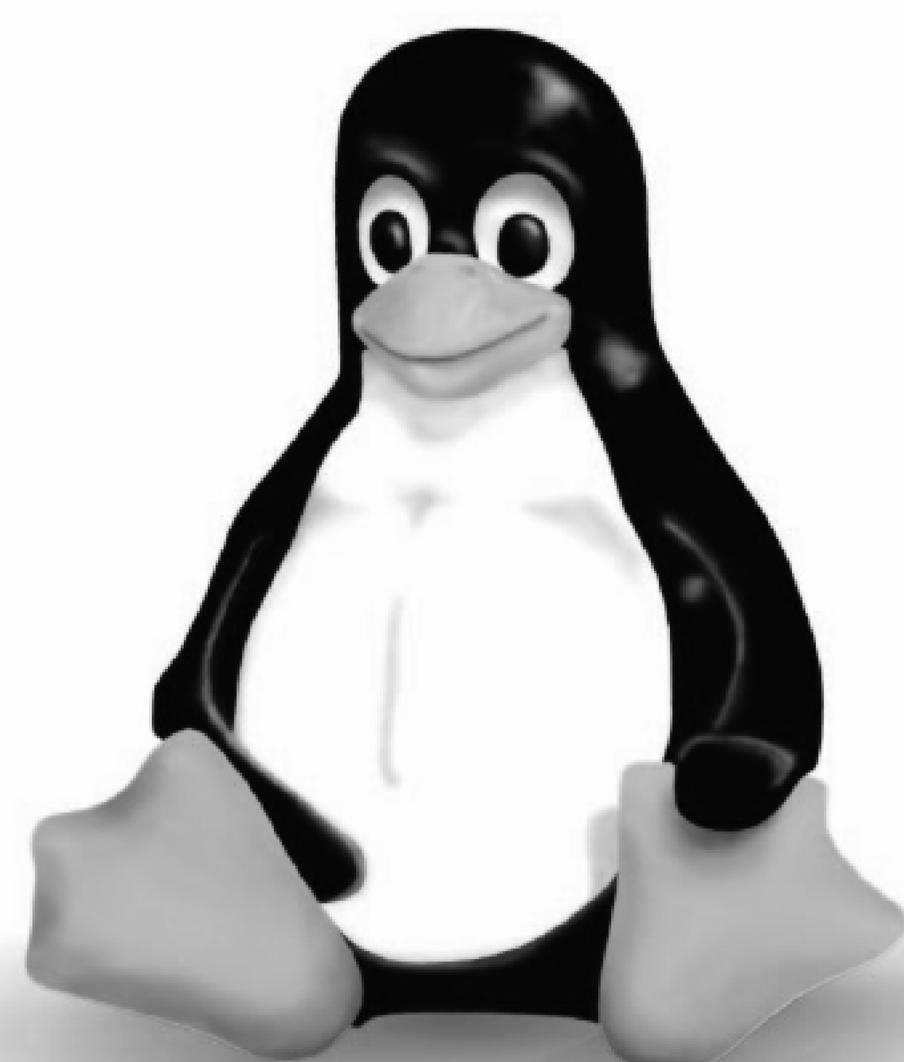
Linux C 编程从基础到实践

·程国钢　张玉兰　编著·



提供本书资源文件下载

清华大学出版社



Linux C 编程从基础到实践

·程国钢 张玉兰 编著·

清华大学出版社
北 京

内 容 简 介

Linux 是在 Unix 的基础上发展起来的一套可以免费使用和自由传播的操作系统，从 1991 年问世至今已经走过了 20 多年的历史，Linux 从一个简单架构的系统内核发展到了现在结构完整、功能丰富的多版本用户系统，已经成为现今世界上最流行的操作系统之一，其不仅能在 PC 机和服务器上运行，随着嵌入式系统的发展，Linux 操作系统已经被广泛应用于各种场合。

本书共 13 章，可以分为 6 个部分，基于 Ubuntu 12.04 由浅入深地介绍了在 Linux 下使用 C 语言进行系统开发的基础知识，包括文件和流操作、进程/线程的操作和同步、网络编程、图形界面编程等，并给出了大量实例，同时在本书可下载资源中还搭配了 10 多个小时的相应视频讲解，以帮助读者完成从入门到进阶的提升。

本书既有 Linux 的基础知识介绍，又包含了丰富的应用实例，适合有 C 语言基础和 Linux 操作系统基础的工程师学习，以及高等院校计算机相关专业的学生和其他爱好者阅读。

图书在版编目（CIP）数据

Linux C 编程从基础到实践 / 程国钢，张玉兰编著. －北京：清华大学出版社，2015

ISBN 978-7-302-39725-0

I. ①L… II. ①程… ②张… III. ①Linux 操作系统－程序设计②C 语言－程序设计 IV. ①TP316.89②TP312

中国版本图书馆 CIP 数据核字（2015）第 065962 号

责任编辑：夏非彼
责任校对：闫秀华
责任印制：
出版发行：清华大学出版社　　**地　址：**北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　编：**100084
社 总 机：010-62770175　　**邮　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：
经　销：全国新华书店
开　本：190mm×260mm　**印　张：**38.75　**字　数：**992 千字
版　次：2015 年 6 月第 1 版　**印　次：**2015 年 6 月第 1 次印刷
印　数：1~3000
定　价：89.00 元

产品编号：063031-01

前　言

行业背景

Linux 是在 Unix 的基础上发展起来的一套可以免费使用和自由传播的操作系统，从 1991 年问世至今已经走过了 20 多年的历史，Linux 从一个简单架构的系统内核发展到了现在结构完整、功能丰富的多版本用户系统，已经成为现今世界上最流行的操作系统之一，其不仅能在 PC 机和服务器上运行，随着嵌入式系统的发展，Linux 操作系统已经被广泛应用于各种场合。

关于本书

本书共 13 章，可以分为 6 个部分，基于 Ubuntu 12.04 由浅入深地介绍了在 Linux 下使用 C 语言进行系统开发的基础知识，包括文件和流操作、进程/线程的操作和同步、网络编程、图形界面编程等，对每个部分及章节的内容说明如下。

- 第 1 部分（包括第 1 章和第 2 章）：介绍 Linux 操作系统中的基本概念和使用方法，以及在 Linux 中进行 C 语言编程的方法、在 Linux 中应用 C 语言代码的技巧。
- 第 2 部分（包括第 3～6 章）：介绍如何在 Linux 操作系统中使用 C 语言对文件和流进行操作，涉及基础文件操作、目录文件操作、文件系统、文件权限以及流 I/O 的操作方法。
- 第 3 部分（包括第 7～10 章）：介绍在 Linux 操作系统中进程、线程的相关知识，以及如何使用 C 语言对进程和线程进行相应的操作，如何使用信号、管道等方法进行进程之间的同步和数据交互。
- 第 4 部分（第 11 章）：介绍如何在 Linux 下使用 C 语言进行网络编程，包括基础网络信息转换，以及 TCP/IP 和 UDP 协议编程。
- 第 5 部分（第 12 章）：介绍如何在 Linux 下使用 GTK+进行图形界面的设计，以及如何设计一个窗口。
- 第 6 部分（第 13 章）：介绍两个 Linux 下的综合应用实例，一个是基于串口和文件编程的风力数据采集系统，另一个是俄罗斯方块游戏。

本书特色

- 循序渐进，由浅入深：从最基本的 Linux 起源和分类开始，逐步介绍在 Linux 中编写 C 语言的方法、Linux 的文件、流、进程、线程、网络编程、图形编程等知识，以及如何使用 C 语言进行相应的实现和操作。
- 实例丰富：本书对于介绍的相应知识都基于 Linux 的 Ubuntu 发行版给出了大量的实例。

- 辅助视频：本书录制了长达 10 多个小时的视频，由浅入深地介绍了 Linux 的基础使用方法以及相应的 C 语言函数使用方法。

下载资源

本书配套资源的下载地址：http://pan.baidu.com/s/1kTHxOFl，若下载有问题，请发送电子邮件至 booksaga@126.com，邮件标题为“求下载资源，Linux C 编程从基础到实践”。

作者介绍

本书由程国钢、张玉兰编写，同时，参与本书编写工作的还有吕平、高克臻、张云霞、张璐、许小荣、王冬、王龙、张银芳、陈可汤、陈作聪、苏静、周艳丽、祁招娣、张秀梅、李爽、卿前华、王文婷、肖岳平、肖斌、蔡娜等人。由于时间仓促，程序较多，受学识水平所限，不足之处在所难免，请广大读者给予批评指正。

编者

2015 年 3 月

目　录

第 1 章　Linux 使用基础 1
1.1　Linux 发展大事记 1
1.2　Linux 的特点 4
1.3　Linux 的几个相关术语 5
1.3.1　GNU 5
1.3.2　GPL 5
1.3.3　POSIX 6
1.3.4　ISO C 6
1.4　Linux 的体系结构 6
1.4.1　Linux 的内核 6
1.4.2　Linux 的命令解释层 7
1.4.3　Linux 的文件系统 7
1.4.4　Linux 的应用软件 8
1.5　Linux 的内核版本和发行版本 8
1.5.1　Linux 的内核版本 9
1.5.2　Linux 的发行版本 10
1.6　Linux 的包管理 14
1.7　Linux 的人机交互 16
1.7.1　图形界面 16
1.7.2　shell 18
1.8　shell 的使用 22
1.8.1　shell 命令的标准格式 22
1.8.2　shell 的通配符 23
1.8.3　shell 中的引号 24
1.8.4　shell 中的注释符 26
1.9　Linux 的常用命令 26
1.9.1　目录操作命令 26
1.9.2　文件操作命令 28
1.9.3　其他命令 35
1.10　本章习题 41

第 2 章　在 Linux 下进行 C 语言开发 42
2.1　C 语言的特点和开发流程 42
2.2　Linux 下的 C 语言开发工具 43
2.3　Linux C 语言的代码编辑工具 44
2.3.1　vim 44
2.3.2　Emacs 52
2.3.3　gedit 52
2.3.4　在 Linux 中编辑 C 语言代码文件的应用实例 53
2.4　Linux C 语言的编译器 gcc 55
2.4.1　gcc 的安装和配置 55
2.4.2　gcc 对 C 语言的处理过程 56
2.4.3　gcc 的基础使用方法 58
2.4.4　gcc 的应用实例 59
2.5　Linux C 语言的调试工具 gdb 60
2.5.1　gdb 的基础使用 61
2.5.2　gdb 运行模式的选择 63
2.5.3　gdb 应用实例 63
2.6　Linux C 语言的项目管理工具 make 64
2.6.1　make 项目管理器的基础 64
2.6.2　make 项目管理器的使用 69
2.6.3　make 项目管理器的应用实例 70
2.7　Linux 中的 C 语言应用代码 73
2.7.1　C 语言代码的运行机制 73
2.7.2　C 语言代码的程序存储空间 75
2.7.3　C 语言代码的 main 函数和参数 76
2.7.4　C 语言代码的出错处理 78
2.7.5　C 语言代码的标准输入和输出函数 83
2.7.6　C 语言代码的时间处理 85
2.7.7　C 语言代码的分配机制 89
2.7.8　C 语言代码的系统调用和库函数 90
2.8　本章习题 91
第 3 章　Linux 文件的基础操作 92
3.1　Linux 的文件 92
3.1.1　Linux 的文件类型 93
3.1.2　Linux 的文件结构和文件描述符 95
3.2　Linux 的文件基础操作 96
3.2.1　打开和关闭文件 96

3.2.2 创建文件 101
3.2.3 将数据写入文件 103
3.2.4 在文件中进行定位操作 113
3.2.5 从文件中读出数据 119
3.3 文件基础操作的综合应用——定时创建文件并且写入数据 122
3.3.1 综合应用的需求说明和分析 123
3.3.2 秒定时的实现 123
3.3.3 将当前时间信息写入文件 126
3.3.4 使用时间信息作为文件名 129
3.4 本章习题 134
第 4 章 Linux 的目录文件操作 135
4.1 目录文件的基础操作 135
4.1.1 目录文件的创建和删除 135
4.1.2 目录文件的打开、关闭和读取 140
4.1.3 当前工作目录路径 147
4.2 目录文件的综合应用——定时创建目录和文件 150
4.2.1 综合应用的需求分析 150
4.2.2 使用时间信息生成目录和 文件 150
4.2.3 定时创建目录和文件 153
4.3 本章习题 156
第 5 章 Linux 的文件系统和文件属性 157
5.1 Linux 的文件系统和文件属性基础 157
5.1.1 Linux 的文件系统 158
5.1.2 Linux 的文件系统结构 160
5.1.3 Linux 的文件和文件属性 164
5.2 Linux 文件属性的操作方法 166
5.2.1 判断文件类型 166
5.2.2 文件的时间信息 170
5.2.3 文件的权限 174
5.2.4 修改文件的名称 183
5.2.5 删除文件 185
5.3 Linux 的链接文件 187
5.3.1 链接文件基础 187
5.3.2 硬链接操作函数 188
5.3.3 符号链接操作函数 188
5.4 文件的其他操作 190
5.4.1 dup 和 dup2 函数 190

5.4.2 fcntl 函数 192
5.4.3 truncate 和 ftruncate 函数 195
5.5 本章习题 195
第 6 章 Linux 的流 196
6.1 Linux 的流基础 196
6.1.1 流和文件的关系 196
6.1.2 流的结构和操作流程 197
6.1.3 标准流介绍 199
6.2 流的基础操作 199
6.2.1 打开和关闭流 199
6.2.2 读写流 205
6.2.3 流的出错处理 217
6.2.4 流的定位 220
6.2.5 流的缓冲管理 224
6.2.6 流的冲洗 231
6.3 流的格式化输入和输出 231
6.3.1 流的格式化输出 232
6.3.2 流的格式化输入 232
6.3.3 流的格式化参数 233
6.3.4 流的格式化输入输出操作实例 236
6.4 本章习题 241
第 7 章 Linux 的进程 243
7.1 操作系统和进程 243
7.1.1 进程的特点 243
7.1.2 进程和可执行文件（程序）的区别 244
7.2 Linux 的进程基础 244
7.2.1 Linux 进程的基础属性 244
7.2.2 Linux 的进程标识方法 247
7.2.3 Linux 的进程调度 251
7.2.4 Linux 下的进程执行流程 252
7.3 Linux 的进程操作 252
7.3.1 使用 fork 函数来创建进程 252
7.3.2 执行进程 264
7.3.3 使用 vfork 函数创建并且执行进程 271
7.3.4 退出进程 274
7.3.5 销毁进程 279
7.3.6 Linux 的进程操作总结 281

7.4 进程综合应用——使用多个进程创建文件 283
7.5 Linux 的进程组和会话 286
7.6 Linux 进程的其他操作 290
7.6.1 更改进程用户 290
7.6.2 在进程中调用其他应用程序 292
7.6.3 统计进程状态 295
7.6.4 进程的执行时间 297
7.7 本章习题 301
第 8 章 Linux 的信号 302
8.1 Linux 的信号机制 302
8.1.1 信号的工作方式 302
8.1.2 信号的分类和说明 305
8.2 Linux 信号的使用方法 308
8.2.1 注册信号 309
8.2.2 发送信号 314
8.2.3 定时信号 325
8.2.4 退出信号 326
8.3 Linux 的信号集 327
8.4 信号的高级操作 328
8.4.1 信号的阻塞和挂起 328
8.4.2 信号的精确定时 329
8.4.3 可重入函数 331
8.5 本章习题 332
第 9 章 Linux 的进程同步机制 ——管道和 IPC 333
9.1 Linux 的管道 333
9.1.1 管道的基本概念 333
9.1.2 管道的实现方法 334
9.1.3 管道的读写操作规则 335
9.1.4 管道的特点 336
9.2 Linux 的管道操作 336
9.2.1 管道的创建 336
9.2.2 进程的管道通信 338
9.2.3 管道的高级应用 347
9.3 Linux 的命名管道 350
9.3.1 命名管道的基本概念 350
9.3.2 命名管道的工作方式 350
9.4 Linux 的命名管道操作 354

9.4.1 创建命名管道 354
9.4.2 读写命名管道 355
9.4.3 进程使用命名管道通信 356
9.5 Linux 的 System V IPC 机制 359
9.5.1 System V IPC 的标识符和关键字 360
9.5.2 System V IPC 的结构和权限 361
9.5.3 System V IPC 的特点 364
9.6 Linux 的消息队列 365
9.6.1 消息队列基础 365
9.6.2 消息队列的操作 367
9.7 Linux 的信号量 377
9.7.1 信号量的基础 377
9.7.2 信号量的操作 379
9.8 Linux 的共享内存 383
9.8.1 共享内存的基础 383
9.8.2 共享内存的操作 384
9.8.3 共享内存的应用实例 387
9.9 本章习题 390

第 10 章 Linux 的线程 391

10.1 Linux 的线程基础 391
10.1.1 线程的特点 391
10.1.2 控制线程和线程的标识符 392
10.1.3 用户态和核心态线程 393
10.1.4 使用 gcc 编译线程的相关代码 393
10.2 Linux 的线程操作 393
10.2.1 创建线程 394
10.2.2 阻塞和退出线程 397
10.2.3 取消和清理线程 400
10.2.4 分离线程 401
10.3 线程的属性 402
10.3.1 线程属性对象的初始化和销毁函数 403
10.3.2 线程堆栈大小相关函数 403
10.3.3 线程堆栈地址函数 404
10.3.4 线程的分离状态函数 404
10.3.5 线程的作用域函数 404
10.3.6 线程的继承调度函数 405
10.3.7 线程的调度策略函数 405
10.3.8 线程的调度参数函数 405

10.3.9 使用线程的属性 406
10.4 线程的私有数据 407
10.4.1 创建键函数 408
10.4.2 取消键关联函数 408
10.4.3 解决键冲突函数 408
10.4.4 键关联函数 409
10.4.5 线程私有数据地址获取函数 409
10.4.6 使用线程的私有数据 409
10.5 线程的同步 411
10.5.1 使用互斥锁解决线程同步 412
10.5.2 使用条件变量解决线程同步 415
10.6 本章习题 420
第 11 章 Linux 的网络编程 422
11.1 Linux 的网络通信模型 422
11.1.1 OSI 网络模型 422
11.1.2 TCP/IP 协议和其网络模型 423
11.1.3 客户端/服务器结构 426
11.1.4 Linux 的端口和套接字 427
11.1.5 Linux 套接字的结构定义 429
11.2 Linux 的网络基础操作函数 430
11.2.1 字节顺序转换函数族 431
11.2.2 字节操作函数族 432
11.2.3 IP 地址转换函数族 433
11.2.4 域名转换函数族 436
11.3 Linux 的网络套接字操作函数 439
11.4 Linux 的 TCP 编程 450
11.4.1 TCP 基础 451
11.4.2 TCP 的工作流程和应用 452
11.5 Linux 的 UDP 编程 457
11.5.1 UDP 的基础知识 457
11.5.2 UDP 的工作流程和应用 457
11.6 应用实例——获取网络时间 461
11.7 本章习题 468
第 12 章 在 Linux 中进行基础图形编程 469
12.1 Linux 图形编程基础 469
12.2 GTK+的基本使用方法 471
12.2.1 GTK+的常见数据类型 471

12.2.2 GTK+的常见函数前缀 472
12.2.3 一个最简单的 GTK+窗口 472
12.2.4 GTK+的常见基础函数 473
12.2.5 GTK+的构件和容器 477
12.2.6 GTK+的回调函数 478
12.3 在 GTK+中使用简单构件 480
12.3.1 按钮 481
12.3.2 触发按钮 482
12.3.3 复选框 484
12.3.4 单选框 485
12.3.5 组合盒 487
12.3.6 组合表 489
12.3.7 标签 491
12.3.8 输入框 493
12.3.9 箭头 496
12.3.10 标尺 498
12.4 在 GTK+中使用组合构件 499
12.4.1 对话框 499
12.4.2 组合框 501
12.4.3 微调构件 503
12.4.4 日历构件 505
12.4.5 文件选择构件 508
12.4.6 按钮盒 511
12.4.7 框架 513
12.4.8 文本框 514
12.5 设计 GTK+的菜单 517
12.5.1 建立菜单 518
12.5.2 菜单的信号处理 520
12.5.3 工具栏 521
12.6 使用 Glade 界面设计师 523
12.7 本章习题 524
第 13 章 Linux 的 C 语言编程实战 525
13.1 实时风力数据采集仪 PC 机端软件设计 525
13.1.1 实时风力数据采集仪 PC 机端软件的需求分析 525
13.1.2 Linux 下的串口编程基础 526
13.1.3 实时风力数据采集仪 PC 机端软件的代码设计 537
13.1.4 实时风力数据采集仪 PC 机端软件的记录数据 540

13.2 俄罗斯方块游戏设计......540
13.2.1 俄罗斯方块的需求分析......541
13.2.2 GTK+的图形设计进阶......541
13.2.3 俄罗斯方块的代码设计......548
13.2.4 俄罗斯方块游戏的运行......568

附录 习题答案......570

第 1 章　Linux 使用基础

Linux 是在 Unix 操作系统上发展起来的一套可以免费使用和自由传播的操作系统，其已经被广泛应用于普通个人电脑、服务器和嵌入式等场合；在 Linux 上进行 C 语言程序开发，要求开发者必须首先熟悉 Linux 的结构和应用，本章将对 Linux 的基础使用方法进行简要介绍，涉及的内容包括：

- Linux 的发展历史。
- Linux 的特点。
- 常见的 Linux 分类和发行版。
- Linux 的体系结构。
- Linux 的常用操作命令。

1.1　Linux 发展大事记

Linux（读音为：['li:nəks]）从 1991 年第一次出现到如今已经走过了 20 多个年头，其由一个纯教学应用的简单操作系统发展成了一个庞大的、能满足不同应用需求的流行操作系统，随着电子技术的飞速发展，Linux 开始逐步在嵌入式系统中普及并且占领了相当大的市场份额，同时其也出现了类似 iOS 和 Android（安卓）等“同源而不同宗”的新变种。

“Hello everybody out there using minix——I'm doing a (free) operating system”——这是 Linux 之父 Linus Torvalds（李纳斯·托沃兹）在开始创作 Linux 之前发布的宣言，随后一个新的操作系统就诞生了。

1991 年 9 月，Linux 的 0.01 版诞生。

1991 年 10 月，Linux 的 0.02 版本诞生，并且正式被取名为 Linux。

1991 年 11 月，Linux 0.10 版本推出。

1991 年 12 月，Linux 的 0.11 版本随后推出，当 Linux 非常接近于一种稳定可靠的系统时，Linux 决定将 0.13 版本改称为 0.95 版本。

1992 年 4 月，第一个 Linux 新闻组 comp.os.linux 诞生，同时 Linux 的 0.95 版首次可以运行 X-Window。

1992 年 10 月，第一个可以安装的 Linux 版本 SLS（Softlanding Linux System）诞生，它由 Peter MacDonald 推出，其安装界面如图 1.1 所示。

1992 年~1994 年期间，三个在随后的岁月中给 Linux 带来深厚影响力的发行版（关于“发行版”的介绍请参考第 1.5 节）Slackware、Red Hat（红帽）和 Debian（蝶变）都出现了。



图 1.1　第一个可安装的 Linux 版本 SLS 安装界面

1994 年 3 月，终于出现了带有独立宣言意味的 Linux 1.0.0 版本，共 176.250 行代码，此时 Linux 已经成为了一个功能完备的操作系统，其内核写得紧凑高效，可以充分发挥硬件的性能，在 4MB 内存的 80386 机器上也表现得非常好，同时其还支持 TCP/IP 栈和 X 窗口系统，图 1.2 是一个当时的 Linux 运行界面截图。此时 Linux 的三大著名发行版包括了 Debian、SUSE（由 Slackware 发展而来）、Red Hat。



图 1.2　Linux 的 1.0.0 版本运行截图

1995 年~2000 年 Linux 得到了飞速的发展，从以上三个大的发行版中分裂出了许多其他优秀的发行版，其中就有基于 Red Hat 发展而来的中科红旗（Red Flag）。在这几年中最重要的事件莫过于 KDE 和 GNOME 的发布，前者于 1998 年发布 1.0 版本，被 Mandrake 发行版第一个采用，其早期界面如图 1.3 所示；而后者于 1999 年发布了第一版，随后被 Red Hat 采用，图 1.4 是 Red Hat 上运行的 GNOME。

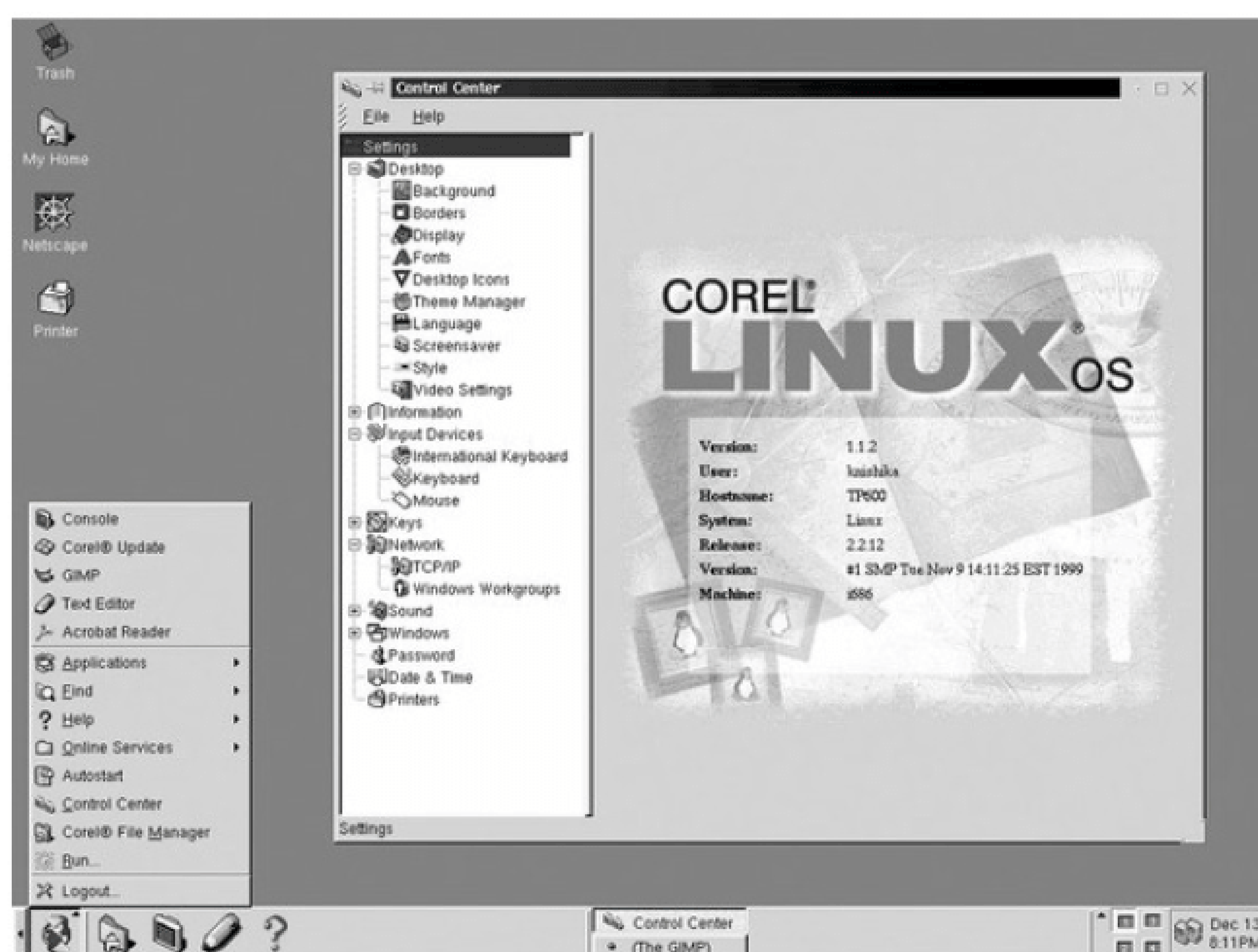


图 1.3　早期 KDE 的运行界面

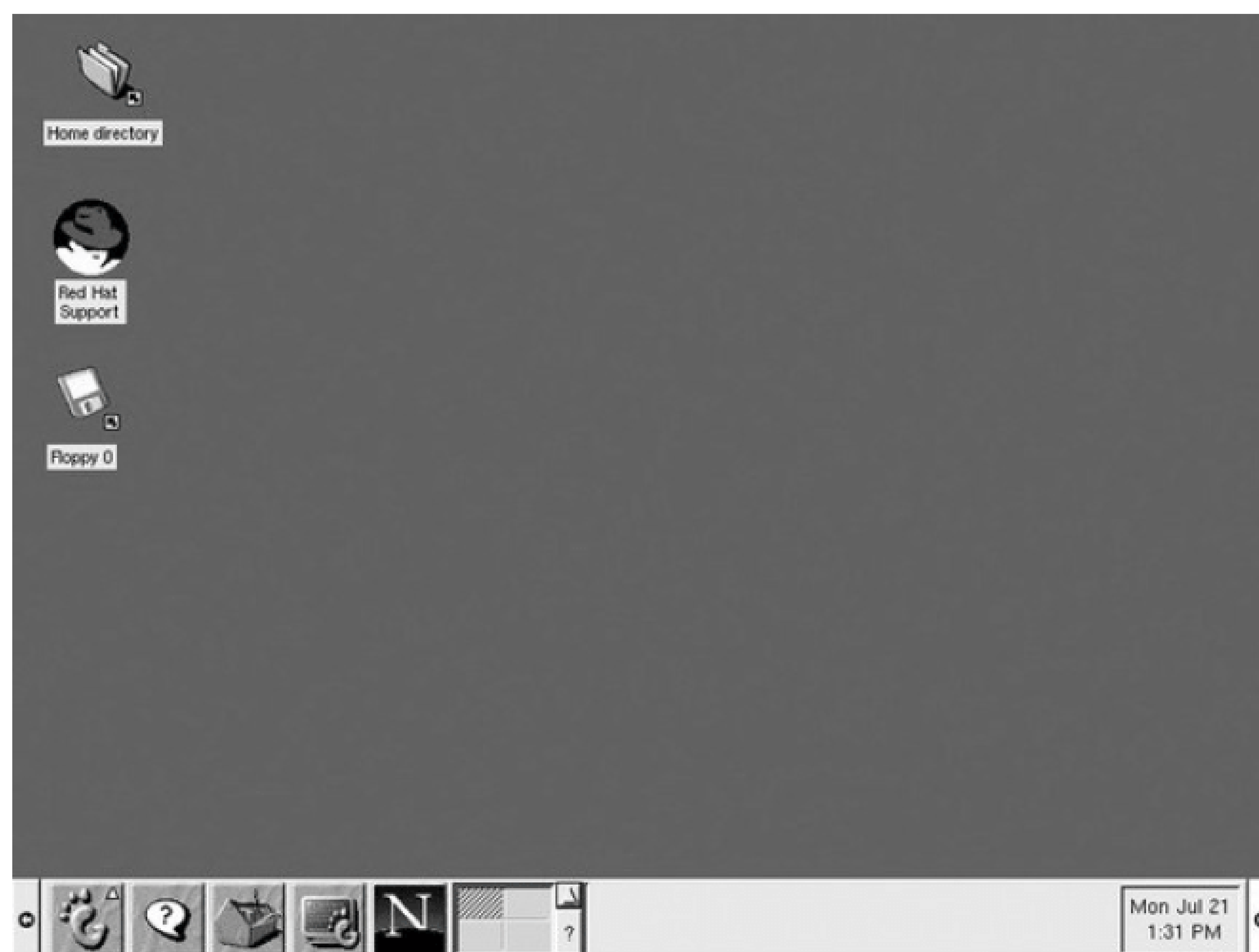


图 1.4　早期 GNOME 的运行界面

2000 年~2005 年，此时的 Linux 在众多开发者的共同努力下得以完善，2.4 版本的 Linux 内核提供了对 USB、PC 卡、ISA、蓝牙、RAID 和 EXT3 文件系统等的支持；2.6 版本的 Linux 内核则进一步提供了对 PAE、64 位处理器、16TB 大容量存储器以及 EXT4 文件系统等的支持。2000 年 Linux 基金会成立，开始赞助 Linux 系统的相关开发工作，并且致力于维护 Linux 内在的自由、合作的核心价值观。在这些年中 Linux 操作系统还出现了一种新的发行版形式——Live（使用），这种发行版支持从光盘直接运行 Linux 操作系统，可以给不熟悉 Linux 的用户最快捷的体验。同样在这些年中包括 Red Hat 在内的大量商业 Linux 公司开始架构和发布不同的商业 Linux 版本，大大促进了 Linux 操作系统的发展。在 2004 年 10 月 20 日，截至目前（2014 年）最为流行的桌面 Linux 发行版 Ubuntu（乌班图）发布了第一个版本 4.10。

2006 年~2014 年，在这段时间内 Linux 的发行版呈现了爆炸式的增强（参考表 1.1），KDE 发布了 KDE 4.2，GNOME 则发布了 GNOME 3，一个全新的桌面环境 Unity 在 Ubuntu 的 11.04 发行

版上出现，而 Linux 内核也发布到了 3.13.3（2014 年 2 月 13 日）。在这几年中的另外一个突破则是 Linux 被大量的移植到基于 ARM 等处理器的嵌入式系统中，而基于 Linux 内核的移动端商用操作系统 Android（安卓）也在 2009 年 9 月发布。

表 1.1　Linux 操作系统的发行版

原始版本	最新发行版		
Debian	Debian	Ubuntu	Linux Mint
	Knoppix	MEPIS	Sidux
	CrunchBang Linux	Chromium OS	Google Chrome OS
Red Hat	Red Hat Enterprise Linux	Fedora	CentOS
	Scientific Linux	Oracle Linux	
Mandriva	Mandriva Linux	PCLinuxOS	Unity Linux
	Mageia		
Gentoo	Gentoo Linux	Sabayon Linux	Calculate Linux
	Funtoo Linux		
Slackware	Slackware	Zenwalk	VectorLinux
其他	SUSE	Arch Linux	Puppy Linux
	Damn Small Linux	MeeGo	SliTaz
	Tizen	StartOS	

1.2　Linux 的特点

作为一个参考 Unix 设计并且得到了广泛应用的操作系统，Linux 在具有 Unix 全部特性的基础上也拥有自己的特性：

- 开发性和自由性：Linux 操作系统完全兼容 POSIX 1.0（POSIX 是基于 Unix 的第一个操作系统国际标准），并且遵循开放系统互联（OSI）国际标准，凡是遵循国际标准所开发的硬件和软件，都能彼此兼容，可方便地实现互联，其是作为开放源代码的自由软件代表，无数的开发者参与了其开发工作，并且可以根据自己的兴趣和灵感对其进行改变。
- 支持多用户和多任务：Linux 操作系统支持多用户，不同的用户可以对属于自己的硬件资源有对应的权限且互不影响，同时 Linux 操作系统还可以同时执行多个任务，并且这些任务之间是相互独立的。
- 提供友好的用户交互界面：Linux 操作系统向用户提供了文本界面和图形用户界面，用户既可以使用 shell 这种基于文本的命令行界面和系统进行交互，也可以使用 X-Window 这种图形界面通过鼠标、菜单、窗口、滚动条等设施和系统进行交互。
- 支持多种文件系统：Linux 操作系统能支持多种文件系统，包括 EXT、EXT2、EXT3、ISOFS、HPFS、MSDOS、UMSDOS、PROC、NFS、XFS、SYSV、MINIX、SMB、UFS、NCP、VFAT、NTFS、AFFS 等。

- 具有设备独立性：Linux 操作系统把所有的外部设备都当做文件，只要安装这些设备对应的驱动程序就可以像使用文件那样进行操作并使用这些设备，而不必知道它们的具体存在形式。

此外 Linux 操作系统还具有丰富的网络功能、可靠的系统安全、文件传输、远程访问、良好的可移植性等特点。

1.3 Linux 的几个相关术语

在 Linux 中，有几个术语是 Linux 下的 C 语言程序员必须了解的，包括 GNU、GPL、POSIX 和 ISO C。

1.3.1 GNU

GNU 是“GNU's Not Unix”的递归缩写，其发音为“Guh-NOO”，原意为非洲牛羚，其最开始由 Richard Stallman 创建于 1983 年，目的是为了实现一个软件丰富且可以自由使用的软件库，因此 GNU 计划可以分别开发不同的操作系统部件。GNU 计划采用了部分当时已经可自由使用的软件，例如 TeX 排版系统和 X Window 视窗系统等，同时也开发了大批其他的自由软件。

1985 年 Richard Stallman 又创立了自由软件基金会（Free Software Foundation）来为 GNU 计划提供技术、法律以及财政支持，尽管 GNU 计划大部分时候是由个人自愿无偿贡献，但 FSF 有时还是会聘请程序员帮助编写。当 GNU 计划开始逐渐获得成功时，一些商业公司开始介入开发和技术支持，当中最著名的就是之后被 Red Hat 兼并的 Cygnus Solutions。

到 1990 年时，GNU 计划已经开发出的软件包括了一个功能强大的编辑器 Emacs、GCC（GNU Compiler Collection，GNU 编译器集合，也是本书所使用的编译器）以及大部分 Unix 系统的程序库和工具，唯一依然没有完成的重要组件就是操作系统的内核。

GNU 包含以下 3 个协议条款。

- GPL：GNU 通用公共许可证（GNU General Public License）。
- LGPL：GNU 较宽松公共许可证（GNU Lesser General Public License），也被称为 GNU Library General Public License（GNU 库通用公共许可证）。
- GFDL：GNU 自由文档许可证（GNU Free Documentation License）。

1.3.2 GPL

GPL 是 General Public License 的缩写，GNU 通用公共授权，其并非由自由软件基金会所发表，亦非使用 GNU 通用公共授权软件的法定发布条款，只有 GNU 通用公共授权英文原文的版本具有此等效力。

GPL 要求在发布软件的同时必须发布源代码，并且允许任何用户能够以源代码的形式将软件复制或者发布给别的用户，如果一个软件使用了遵循 GPL 的任何软件的全部或者一部分，则该软件也必须遵循 GPL。

需要注意的是，GPL 并不是免费软件的代名词，其支持商业化的收费软件。

1.3.3 POSIX

POSIX 是可移植的操作系统（Portable Operating System Interface of Unix）的缩写，其由 IEEE（Institute of Electrical and Electronic Engineering）所开发，由 ANSI 和 ISO 标准化。

POSIX 最初的开发目的是为了提高 Unix 环境下应用程序的可移植性，但是随着其发展，现在 POSIX 并不局限于 Unix 环境，许多其他的操作系统，包括 Linux 和 Windows，也支持 POSIX 的部分或者全部。

1.3.4 ISO C

C 语言是由 Dennis M. Ritchie 在 1973 年设计和实现的，并且在 1978 年通过《The C Programming Language》一书将 C 语言推向全世界。

美国国家标准局（ANSI）在 1988 年 10 月颁布 ANSI 标准 X3.159-1989（即 ANSI C 标准），随后国际标准（ISO）在 1989 年左右采纳 ANSI C 标准，并且将其定义为 ISO/IEC 9899:1990，即为 ISO C。

随着计算机技术的不断进步，ISO C 的版本号也在随之发展，到目前为止最新的 ISO C 版本号是 ISO/IEC 9899:1999，即 C99。

1.4 Linux 的体系结构

一个完整的 Linux 操作系统如图 1.5 所示，由 Linux 内核（Kernel）、命令解释层（shell 等）、文件系统（File Structure）等部分组成。

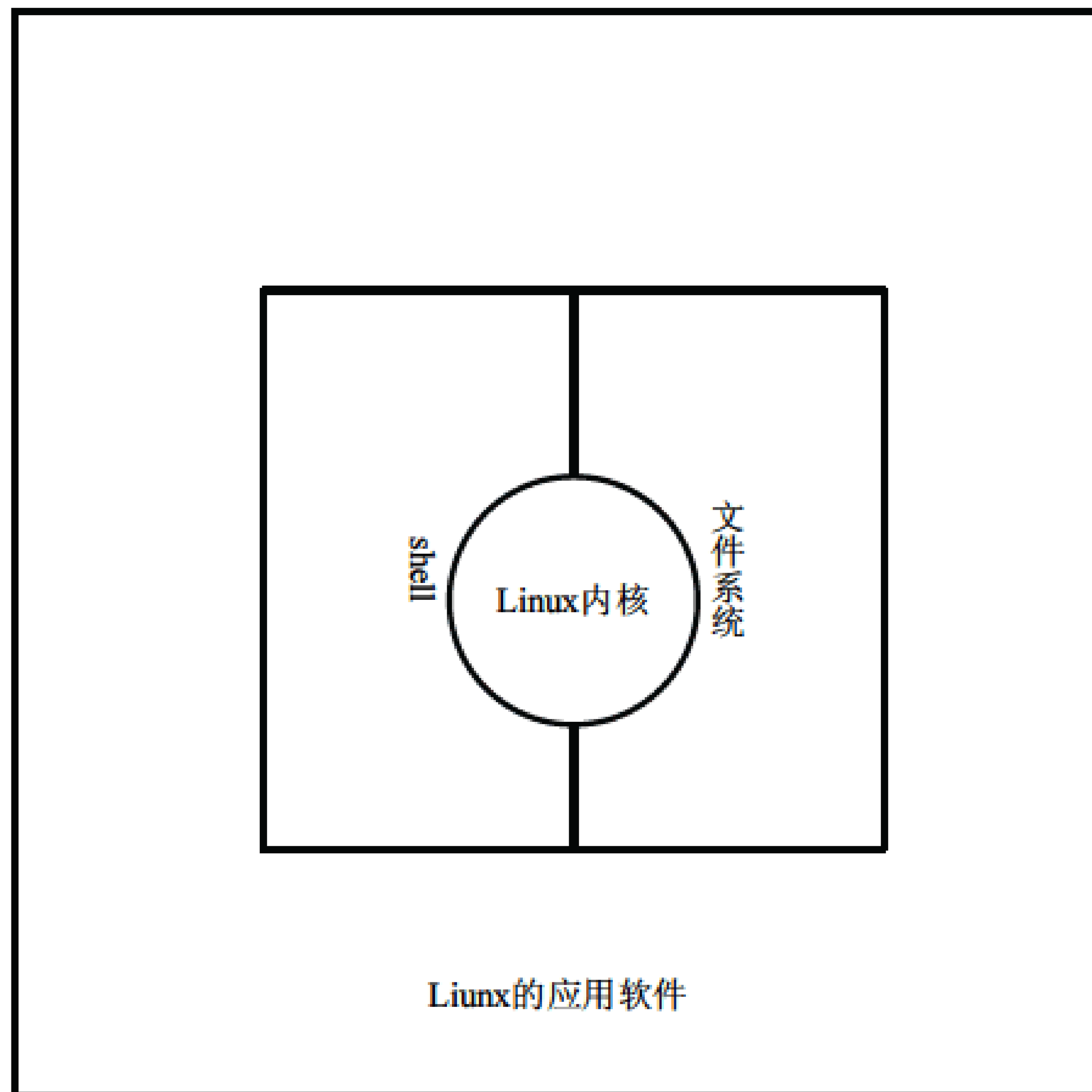


图 1.5 Linux 的体系结构

1.4.1 Linux 的内核

内核（Kernel）是 Linux 操作系统的最基础组成部分，是所有 Linux 操作系统发行版的核心，其以独占的方式执行最底层任务，保证系统正常运行，协调多个并发进程，管理进程使用的内存，使它们相互之间不产生冲突，以满足进程访问磁盘的请求等。

Linux 的内核由如下几个主要模块组成，本书所介绍的 Linux 下的 C 语言编程也是针对这几个组成部分来进行的：

- 文件系统驱动模块。
- 硬件设备驱动模块。
- 内存管理模块。
- 进程管理模块。
- 网络管理模块。

1.4.2 Linux 的命令解释层

命令解释层是操作系统用户和操作系统内核进行信息交互的一种接口，其接收并且解析用户的命令，然后将其送到内核去执行，包括了基于文本的 shell 和基于图形界面的 X-Window 两种。

- shell 解释由用户输入的命令并且把它们送到内核，其也有自己的编程语言用于对命令的编辑，它允许用户编写由 shell 命令组成的程序。shell 编程语言具有普通编程语言的很多特点，例如它也有循环结构和分支控制结构等，利用这种编程语言编写的 shell 程序与其他应用程序具有同样的效果。
- Linux 操作系统同样提供了像 Windows 那样的可视命令输入界面——X-Window 图形用户界面（GUI）。它提供了很多窗口管理器，其操作就像 Windows 一样，有窗口、图标和菜单，所有的管理都是通过鼠标控制。现在比较流行的窗口管理器是 KDE、GNOME 和 Unity。

每个 Linux 操作系统用户都可以拥有他自己的用户界面或 shell，用以满足他们自己专门的 shell 需要。

同 Linux 操作系统本身一样，shell 也有多种不同的版本，目前主流的 shell 包括如下几种：

- 贝尔实验室开发的 Bourne shell。
- GNU 操作系统上默认的 shell，GNU 的 Bourne Again Shell，BASH。
- 在 Bourne Shell 的基础上发展而来的 Korn Shell。
- SUN 公司 shell 的 BSD 版本 shell，C Shell。

1.4.3 Linux 的文件系统

文件系统（File Structure）是指在一个物理设备上的任何文件组织和目录，主要用来存储 Linux 操作系统运行所必需的信息，构成了 Linux 上所有数据的基础。Linux 操作系统中的文件不仅包括普通的文件和目录，另外每个和设备相关的实体也都被映射为一个文件，例如磁盘、终端、打印机、网卡等，这些设备文件称为特殊文件。

Linux 操作系统支持多种文件系统，包括 EXT2、EXT3、VFAT、NTFS、ISO9660、JFFS、YAFFS/YAFFS2、ROMFS 和 NFS 等，为了对各类文件系统进行统一管理，Linux 操作系统引入了虚拟文件系统 VFS（Virtual File System），为各类文件系统提供了一个统一的操作界面和应用编程接口。

Linux 操作系统的文件系统结构如图 1.6 所示，可以分为用户层、内核层和硬件层三个部分。

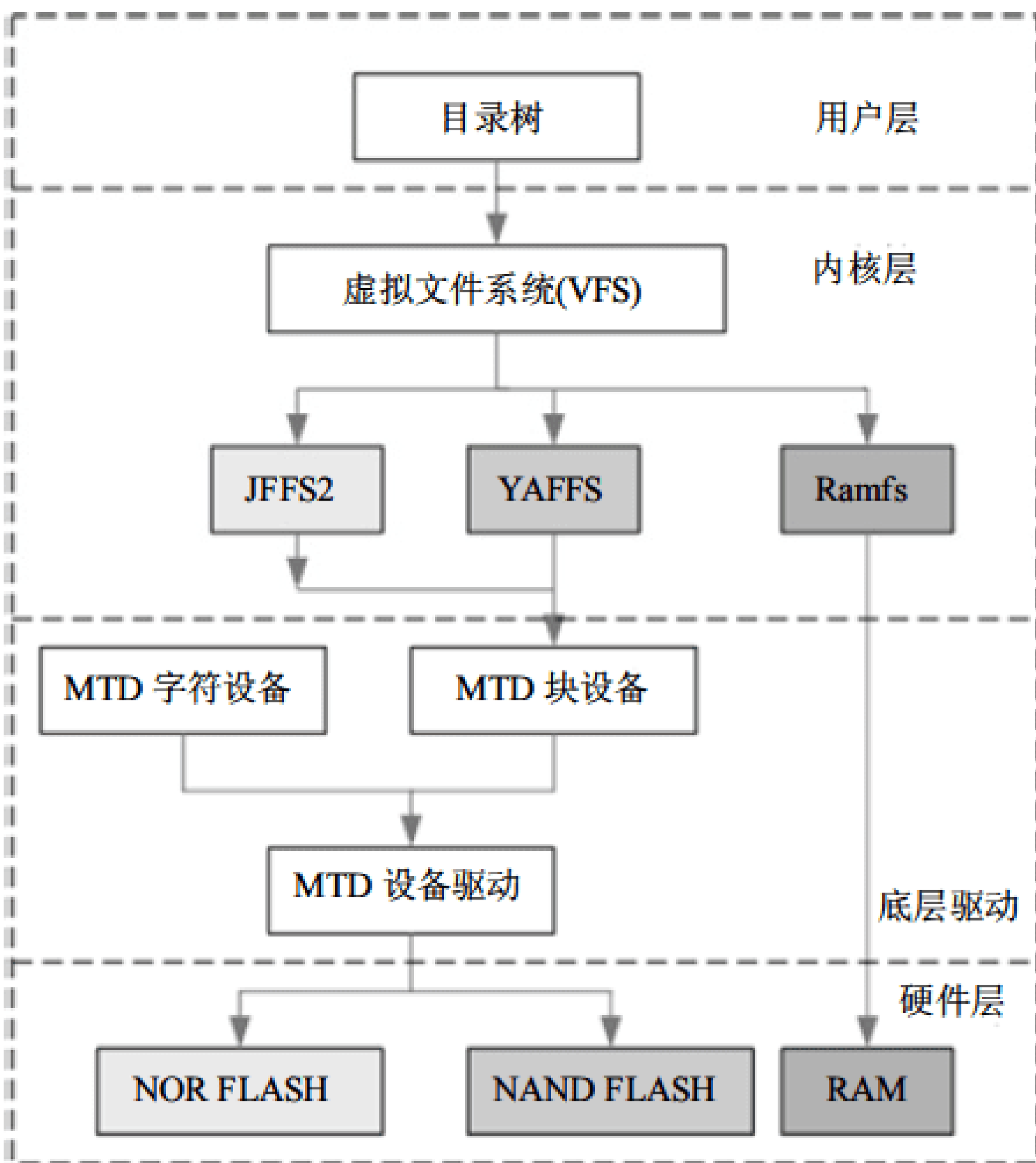


图 1.6　Linux 操作系统的文件系统结构

Linux 文件系统中的文件可以分为普通文件、目录文件和特殊文件三大类：

- 普通文件存放的是数据和程序，也就是二进制流。文件中不包含任何特定的结构。
- 目录文件是一种结构，它允许不同的文件和目录放在一起，类似于 Windows 系统中的文件夹，其中包含的下级目录称为子目录。
- 特殊文件包含多种类型，一般来说，它和不同进程间的通信、计算机和外部设备的通信有关系。

Linux 操作系统中的所有文件都放在一个大的树型结构中。树的根是一个单独的目录，称为根(root)目录，并且用斜杠“/”表示。在根目录下有一些标准的子目录和文件，所谓标准，就是一种传统。在这些子目录下又包含下级子目录和文件，依次类推。图 1.7 是 Linux 操作系统中的目录结构。

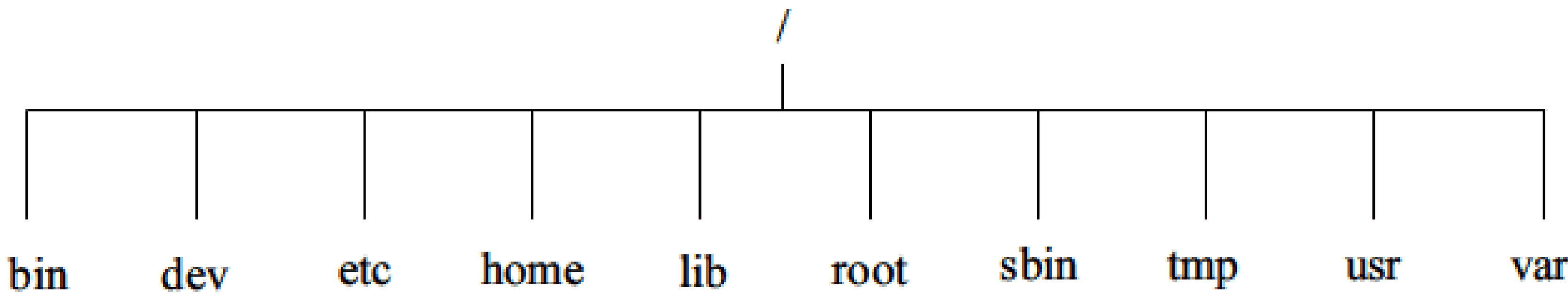


图 1.7　Linux 操作系统的目录结构

1.4.4　Linux 的应用软件

Linux 操作系统发行版通常都会提供一系列应用软件，这些软件包括文本编辑工具和浏览器等，其主要目的是用于增加系统可用性。

1.5　Linux 的内核版本和发行版本

在第 1.4 节中已经介绍过 Linux 操作系统的核心是其内核（Kernel），不同的内核拥有自己的

版本号，发行版（Distribution）即为在某个版本的内核上扩展了其他部件（如桌面环境和应用软件等）得到的一个完整的操作系统。

1.5.1 Linux 的内核版本

Linux 的内核版本号是由 Linus 领导下的开发小组开发出的系统内核版本号，使用形式为 x.y.zz-www 的一组数字来表示，其中 x.y 为 Linux 的主版本号，zz 为次版本号，www 代表发行号（和发行版本号无关）。当内核功能有一个飞跃时，主版本号升级，如 Kernel 2.2、2.4、2.6 等。内核增加了少量补丁时，常常会升级次版本号，如 Kernel 2.6.15、2.6.20 等。此外还有更复杂的版本号系统，如 2.6.20-32 等。通常 y 若为奇数，表示此版本为测试版，系统会有较多 bug，主要用途是提供给用户测试。随着每一次系统小 bug 的修正，zz 都会增加。

Linux 用来支持各种体系结构的源代码包含大约 4500 个 C 语言程序，存放在 270 个左右的子目录下，总共大约包含 200 万行代码，大概占用 58MB 磁盘空间，其文件结构如图 1.8 所示，这里使用的 Linux 系统是 2.6 的内核版本，与其他版本可能会有所差异。

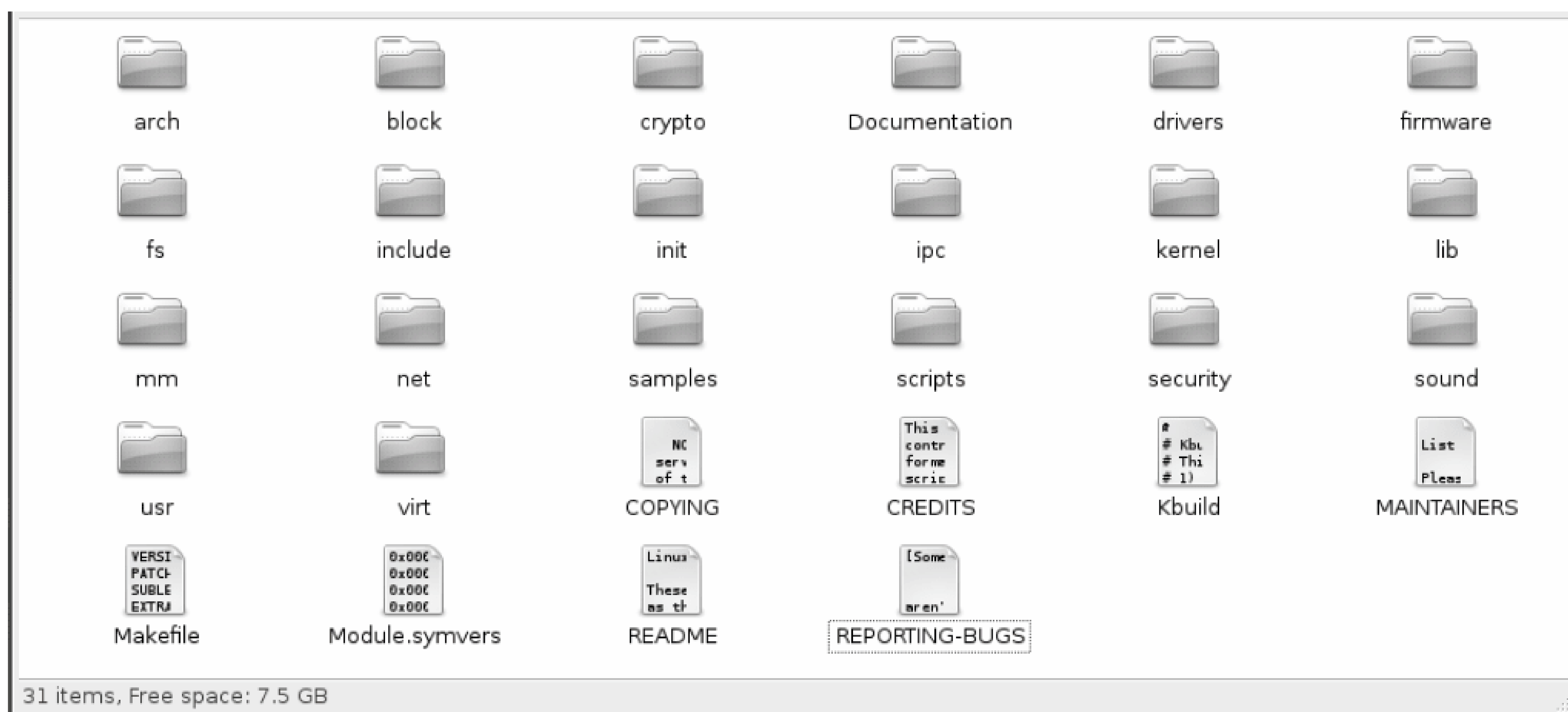


图 1.8 Linux 操作系统的内核文件结构

在 Linux 的内核中可以看到 Linux 的具体管理方法和结构组织，如进程管理、内存管理、文件系统等，与内核源码的各个目录都是对应的，例如有关驱动的内容，内核中都组织到“drive”目录中去，有关网络的代码都集中组织到“net”中。当然，有的目录包含多个部分的内容，对各个目录的内容组成说明如下。

- arch：该目录包括了所有和体系结构相关的核心代码，它下面的每一个子目录都代表一种 Linux 支持的体系结构。
- include：该目录包括编译核心所需要的大部分头文件，例如与平台无关的头文件位于 include/linux 子目录下。
- init：该目录包含核心的初始化代码（不是系统的引导代码），有 main.c 和 Version.c 两个文件。
- mm：该目录包含了所有的内存管理代码，与具体硬件体系结构相关的内存管理代码位

于 arch/*/mm 目录下。

- drivers：该目录中是系统中所有的设备驱动程序，其又进一步划分成几类设备驱动，每一种都有对应的子目录，如声卡的驱动对应于 drivers/sound。
- ipc：该目录包含了核心进程间的通信代码。
- modules：该目录用于存放已建好的、可动态加载的模块。
- fs：该目录存放 Linux 支持的文件系统代码。不同的文件系统有不同的子目录对应，如 ext3 文件系统对应的就是 ext3 子目录。
- Kernel：Kernel 内核管理的核心代码被放在这里，同时与处理器结构相关的代码都放在 arch/*/kernel 目录下。
- net：该目录里是核心的网络部分代码，其每个子目录对应于网络的一个方面。
- lib：该目录包含了核心的库代码，不过与处理器结构相关的库代码被放在 arch/*/lib/目录下。
- scripts：该目录包含用于配置核心的脚本文件。
- documentation：该目录下的文档是对每个目录作用的具体说明。一般在每个目录下都有一个 depend 文件和一个 Makefile 文件。这两个文件都是编译时使用的辅助文件。

1.5.2　Linux 的发行版本

一般而言，一个基本的 Linux 只是包含了 Linux 核心（Kernel）和 GNU 软件的一些基本的系统软件和实用工具（Utilities），这样一个操作系统仅仅能够让那些 Linux 专家完成一些很基本的系统管理任务，若要满足普通用户的办公或基于视窗的应用开发等需要，则还需要在系统中加入 GNOME、KDE 等桌面环境，以及相应的办公应用软件（如 Office）等。因此一些组织或厂家将 Linux 系统内核与 GNU 软件（系统软件和工具）整合起来，并提供一些安装界面和系统设定与管理工具，这样就构成了一个发行套件，例如最常见的 Ubuntu、Fedora 等。实际上发行套件就是 Linux 的一个大软件包而已，通常包括 C 语言及 C++的编译器、Perl 脚本解释程序、shell 命令解释器、图形用户界面以及众多的应用程序等。相对于内核版本，发行套件的版本号随发布者的不同而不同，与系统内核的版本号是相对独立的，因此把 Ubuntu、Fedora 等直接说成是 Linux 是不确切的，它们是 Linux 的发行版本，更确切地说，应该叫做“以 Linux 为核心的操作系统软件包”。根据 GPL 准则，这些发行版本虽然都源自一个内核，并都有自己各自的贡献，但都没有自己的版权。Linux 的各个发行版本，都是由 Linus 主导开发并发布的同一个 Linux 内核，因此在内核层不存在兼容性问题。至于每个版本都不一样的感觉，只是在发行版本的最外层才有所体现，而绝不是本身，也不是内核不统一或不兼容。

许多商业公司在 Linux 的内核上根据自己的需求和特点制定了不同的发行版，最常用的发行版有 Fedora、Ubuntu、SUSE Linux。

1. Fedora

Fedora 是由 Fedora 基金会管理和控制，并得到 Red Hat 公司支持的一个 Linux 发行版，其是一个独立的操作系统，支持硬件体系包括 x86、x86_64 和 PowerPC 等。

2003 年，Red Hat 公司宣布不再推出个人使用的发行版本并转向商业版本的开发，同时 Red Hat

公司也将原来的 Red Hat Linux 开发计划和 Fedora 计划重新整合成一个新的 Fedora 项目，它是在 Red Hat Linux 9 的基础上加以改进而成的。Fedora 项目预计每年将会发行 2~3 个版本。

2003 年 11 月首个发行版本 Fedora Core 1 正式推出，它更新了部分套件，但是并没有完善 Red Hat 的部分相关功能。

2004 年 5 月，Fedora Core 2 正式发布，其版本代码为 Tettnang。这一版本除采用 Xorg X11 取代 XFree86 外，还加入了 IIIMF、SELinux 等许多新技术，并且在开放性原始代码社区的支持下修正了许多套件的错误。同年 11 月，Fedora Core 3 正式发布，其版本代码为 Heidelberg。这一版本采用了 Xorg 6.8.1、GNOME 2.8 和 KDE 3.3.0。

2005 年 6 月，Fedora Core 4 正式发布，版本代码为 Stentz。这一版本采用了 GNOME 2.10、KDE 3.4.0、GCC 4.0 和 PHP 5.0。此外还添加了对 PowerPC 的支持。

2006 年 3 月，Fedora Core 5 正式发布，版本代码为 Bordeaux。GNOME 桌面基于 2.14 发布，KDE 桌面是 3.5 的一般版本，它首次包含对 Mono，以及众多 Mono 应用程序的支持，以 SCIM 语言输入框架取代了过去使用的 IIIMF 系统。同年 10 月，Fedora Core 6 正式发布。

2007 年的 6 月和 11 月，分别推出 Fedora Core 7 和新版本的 Fedora 8。

2013 年 12 月 17 日发布了 Fedora 20，其运行界面如图 1.9 所示，提供了 32 位和 64 位的包括 GNOME、KDE、LXDE、Xfce 在内的多个桌面环境版本，同时也提供了支持 ARM 处理器的 Fedora 20 ARM 架构专用版（只支持 32 位）。用于 ARM 处理器的 Fedora 提供了两种版本：一种用于需要 VFAT 分区的平台（BeagleBone Black），另一种用于从 EXT 分区引导的设备（Trimslice），每一种又都可在 MATE、KDE、Xfce 等桌面环境中进行选择，同时还有不带桌面环境的最小化版本。

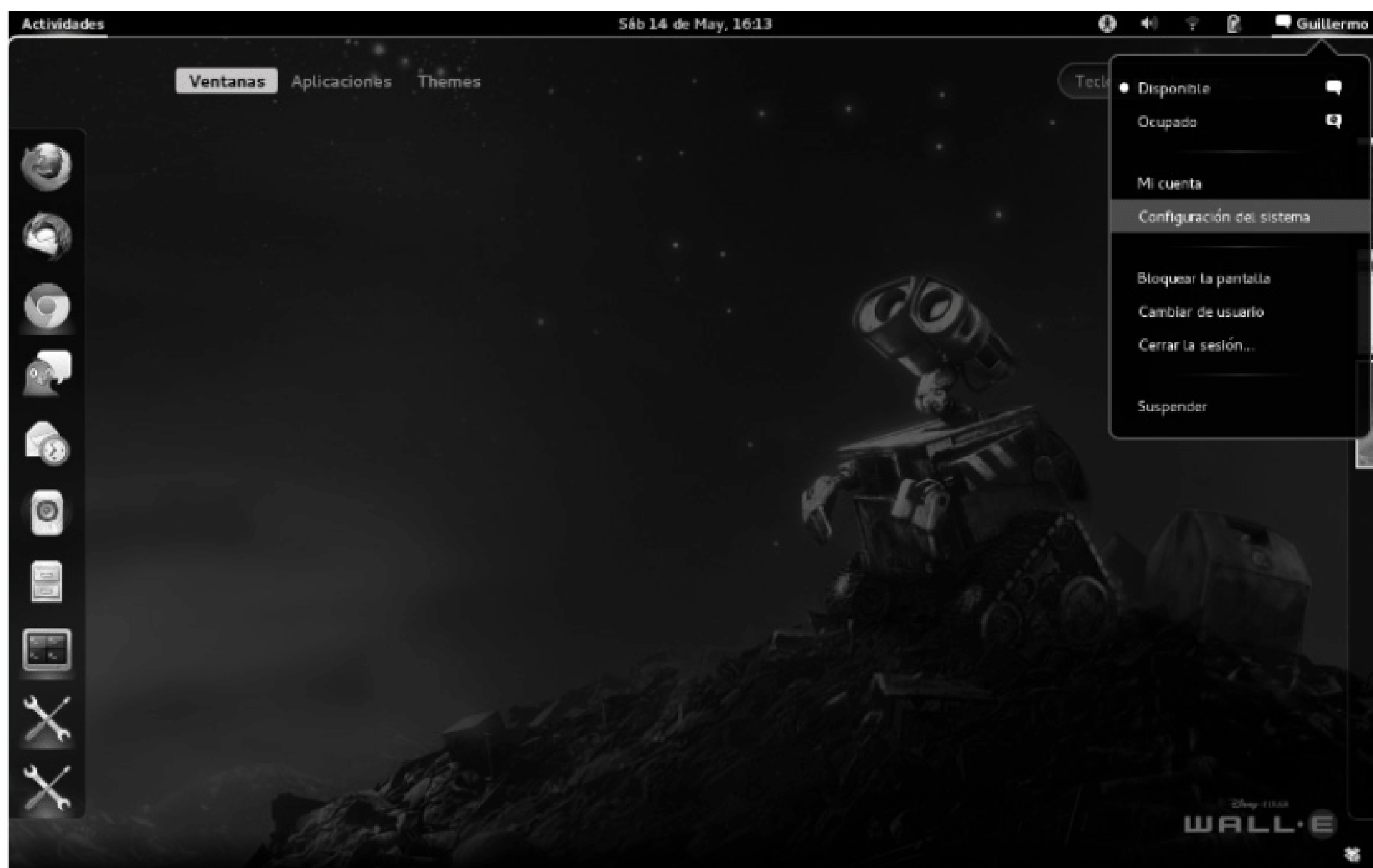


图 1.9　Fedora 20 的运行界面

Fedora 是基于 Linux 环境的、对外开放的、创新的和具有前瞻性的操作系统平台。Fedora 允许任何用户自由地使用、修改并重新发布，拥有熟练庞大的用户群并具有强大的社群开发能力，社

群成员提供并维护自由开放的源代码和开放的标准。Fedora 项目由 Fedora 基金会管理和控制，得到了 Red Hat Inc 的支持。其可运行的体系结构包括 x86、x86-64、PowerPC Fedora Core、ARM 等，它是众多 Linux 发行套件之一。

最新版本的 Fedora 20 具有以下特性：

- 100%的自由开源。
- 千款免费应用。
- 没有病毒和间谍软件。
- 存在一个由来自全球的社区贡献者创建并且有合适个人的本地化站点的全球社区。

Fedora 20 还内置了一些和协作性、娱乐及媒体、创新、办公/生产力相关的应用，例如其提供了 Evolution 邮件及日历套件、Empathy 视频及文字聊天工具、Totem 电影播放机、Gimp 图像编辑器、Scribus 多页排版工具、Audacity 音频编辑器、LibreOffice、Gnote 便签等。

2. Ubuntu

Ubuntu（乌班图）是基于 Debian GNU/Linux，支持 x86、amd64（即 x64）和 ARM 架构，由全球化的专业开发团队（Canonical Ltd）打造的开源 GNU/Linux 操作系统。

Ubuntu 每 6 个月发布一个新版本，而每个版本都有代号和版本号，其中有 LTS 是长期支持版。版本号基于发布日期，例如最新的版本 14.10 是在 2014 年 10 月发行的。

在 Ubuntu 的发展历史上有两个很重要的版本：

- 8.04 LTS 版，在该版本中提供了 WUBI（Windows Ubuntu-Based Installer）安装方式，其支持用户在 Windows 中以安装普通应用软件的方法安装和卸载 Ubuntu，其在不需要改变分区设置，不需要修改启动文件，不会给 Windows 带来任何改变的同时提供了完整的硬件接入。
- 11.04 版，在该版本中采用了 Unity 作为默认的桌面环境，Unity 是一个由 Canonical 公司开发的基于 GNOME 的桌面环境，其目的是更有效地利用显示器的屏幕尺寸并且消耗更少的系统资源，例如将一些快捷方式放在左侧，如图 1.10 所示是 Unity 的运行界面。



图 1.10　Unity 桌面环境

Ubuntu 作为 Linux 发行版的最大特点是比其他发行版本的界面更加友好，同时有更好、更加稳定的技术支持和较快的更新速度，方便对计算机不熟悉的用户使用。

Ubuntu 具有以下一些特点：

- 桌面环境集成了一些常用社交网站、音乐站点，并且支持大量的邮件和新闻服务。
- 支持从远程主机登录（需要设置远程登录账号）。
- Unity Dash 可以提供 Amazon 网络搜索结果。
- 附加驱动整合到了软件源。
- 提供了对普通桌面、手机、平板电脑、电视、服务器等不同应用的多种版本。


注 意

本书使用的 Linux 发行版是 Ubuntu 12.04.4 LTS，其内核版本是 3.8.0-35。

3. SUSE Linux

SUSE 是最早的 Linux 商业发行版，但 SUSE Linux 的使用仍然是免费的。其第一个发行版在 1994 年推出，目前 SUSE 系列主要有个人版和企业版，它们各有自己的优点，其主要特性如下。

- 标准化兼容：所有的 SUSE 系列版本都遵守 Linux 的基本标准集（LSB），并得到了认证。在基本标准集里包含了可移植操作系统接口（POSIX）兼容性的测试，使得在兼容系统之间的代码移植更方便。SUSE Linux 的桌面效果如图 1.11 所示。

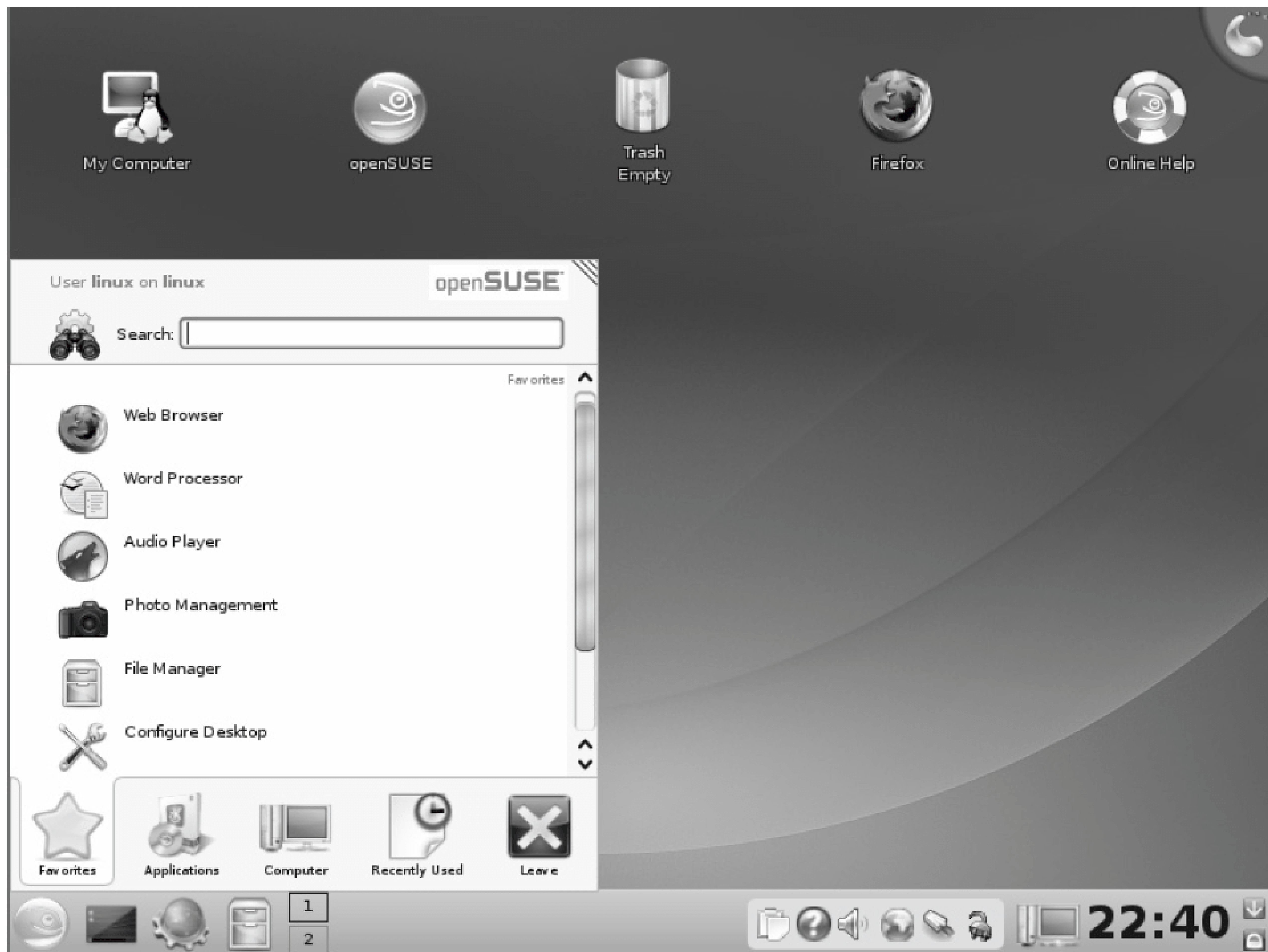


图 1.11　SUSE Linux 的桌面效果

- EAL 认证：EAL 是一个根据国际协约而建立的认证组织，其认证方案与认证方法由通用标准组织提供。2004 年 SLES 8 成功通过了 EAL3+认证，次年 SLES 9 通过了 CAPP/EAL4+的认证。自此以后 SUSE Linux 得到了广泛的接受和认同，加快了其普及的速度。

1.6　Linux 的包管理

Linux 操作系统中所安装的软件通常都是以包的形式存在的，包除了可执行文件之外，其中通常来说还包括了该包的依赖关系、设置文件等。

所谓依赖关系，就是指 Linux 软件运行必须的其他条件，例如软件 A 在运行的时候需要相应的库，如果此时库没有被安装，则软件 A 就不能正常运行，这种情况通常被称为软件 A 的依赖关系没有被满足。

由于在 Linux 中软件和库都非常多，并且一个库可能涉及多个软件，所以 Linux 提供了相应的包管理系统来对这些包进行管理，目前市场占有率最高的两个包管理软件是 RPM 包管理软件和 Deb 包管理软件。

RPM 包管理软件的全称是 Red Hat Package Manager，其是 Red Hat 公司设计的一套包管理软件，其中包括了软件的可执行程序、相关的配置文件等。利用解压缩工具解开 RPM 包即可看到其中的内容，但是如果需要安装 RPM 包，则需要使用相应的包管理工具。

RPM 的包管理工具可以提供包的安装、卸载、查询、打包等功能，在包里面有可执行程序以及相应的安装、依赖关系，以下是几个常用的 RPM 包操作命令：

- rpm -vih file.rpm：安装一个 RPM 包。
- rpm -e file.rpm：卸载一个 RPM 包。
- rpm -qpR file.rpm：查看 RPM 包的依赖关系。
- rpm -q file：查询系统已安装的 RPM 包。

使用 RPM 包管理工具的常见 Linux 发行版包括 Red Hat（Fedora）和 Linux：

- Red Hat（Fedora）：其是美国 Red Hat 公司的产品，是相当成功的一个 Linux 发行版本，也是目前使用最多的 Linux 发行版本。Red Hat 最早由 Bob Young 和 Marc Ewing 在 1995 年创建。原来的 Red Hat 版本早已停止技术支持，目前 Red Hat 的 Linux 分为两个系列，其中一个是由 Red Hat 公司提供收费技术支持和更新的 Red Hat Enterprise Linux 系列；另一个是由社区开发的免费 Fedora Core 系列。Red Hat 因其易于安装而闻名，在很大程度上减轻了用户安装程序的负担，其中 Red Hat 提供的图形界面安装方式非常类似于 Windows 系统的软件安装。
- 红旗 Linux：其是由北京中科红旗软件技术有限公司开发的一系列 Linux 发行版，包括桌面版、工作站版、数据中心服务器版、HA 集群版和红旗嵌入式 Linux 等产品。

和 RPM 包管理系统相对应的是 Deb 包管理系统，Deb 的包也是由源代码包和二进制包组成：

- 源代码包：包括一个描述源代码包的.dsc 文件、一个包含 gzip-tar 归档压缩格式的未经修改源代码的.orig.tar.gz 文件、一个包含对源代码作 Debian 特有修改的.diff.gz 文件．可以使用 dpkg-source 打包和解压 debian 源代码文档。
- 二进制包：以.deb 扩展名来表示，这些文件通常称为 DEB 文件，其中包含可执行文件、文档、配置文件、版权信息及其他一些东西。可以使用 Debian 的 dpkg 工具解包（安装）。

一般而言，用户只和二进制包打交道，只有在某些特殊情况下才会求助于源代码包，Debian 软件包在命名时遵循下列约定:

```
<foo>_<版本号>-<Debian 修订号>.deb
```

Deb 包管理系统同样提供了相应的命令和管理工具用于管理操作，对这些命令和管理工具说明如下（均基于 Ubuntu）。

- apt 命令：用于从源列表（可以是 CD、网络等途径）中下载 Deb 包。
- dpkg 命令：通过数据库来对系统中的软件进行管理，这个数据库位于/var/lib/dpkg 目录下。
- aptitude 命令：提供了一个图形界面来对软件包进行管理，功能较为强大，其甚至可以通过终端远程登录运行，但是该命令工具可能需要安装，其运行界面如图 1.12 所示。



图 1.12 aptitude 命令的运行界面

- synaptic: 这是一个运行在 X Window 环境下的包管理软件，用户可以进行图形化的操作。
- gdebi 和 gdebi-gtk：gdebi 是一个命令行的包管理软件，而 gdebi-gtk 是其对应的图形化版本。
- dselect：为在终端运行的一个图形化软件包，其功能实现类似于 synaptic，但是可以在终端中运行，其运行界面如图 1.13 所示，需要注意的是该命令通常需要 root 权限，如果没有该权限则会导致“只读访问”，通常来说可以使用 sudo dselect 命令来获得相应的权限。

```
ebian "dselect" 软件包操作前端 1.16.1.2 (amd64) 版。

 * 0. 来源[A]    选择所使用的安装来源。
   1. 更新[U]    如果可能的话，更新可用的软件包列表。
   2. 选择[S]    对系统中安装的软件包进行取舍。
   3. 安装[I]    安装并升级之前指定的软件包.
   4. 配置[C]    对所有还未配置的软件包进行配置
   5. 删除[R]    删除不需要的软件包
   6. 退出[Q]    退出 dselect 程序

可以用 ^P 和 ^N、方向键、首字母或者数字键来移动游标;
摁 <回车> 来确认。  用 ^L 来刷屏。

版权所有 (C) 1994-1996 Ian Jackson.
版权所有 (C) 2000,2001 Wichert Akkerman.
This is free software; see the GNU General Public License version 2 or
later for copying conditions. There is NO warranty.

只读访问: 仅仅可以查看被选中的项目!
```

图 1.13　dselect 的运行界面

Debian 和 Ubuntu 这两种最常见的发行版都是使用 Deb 包管理系统。



注　意

实际上还存在其他的包管理系统，在此不再多做叙述，有兴趣的读者可以自行查阅相应的书籍。

1.7　Linux 的人机交互

Linux 通常使用图形操作界面或者 shell 和用户进行交互，前者是一个类似 Windows 的操作界面，而后者则为类似 DOS 的命令行输入反馈界面。

1.7.1　图形界面

几乎所有的 Linux 发行版本中都包含了 GNOME 和 KDE 两种图形操作环境，许多 Linux 操作系统默认的图形操作界面为 GNOME，它除了具有出色的图形环境功能外，还提供了编程接口，允许开发人员按照自己的爱好和需要来设置窗口管理器。

X Window，即 X Windows 图形用户接口，是一种计算机软件系统和网络协议，提供了一个基础的图形用户界面（GUI）和丰富的输入设备能力联网计算机。其中软件编写使用广义的命令集，它创建了一个硬件抽象层，允许设备独立性和重用方案在任何计算机上实现。

Linux 的内核并不像 Windows 一样提供了用户能够使用的图形化界面，而 X Window 即是实现这种功能的应用软件，其可分为 KDE 和和 GNOME 大类。此外 Ubuntu 还提供了 Unity 图形界面。

1. KDE

KDE 是 K 桌面环境（Kool Desktop Environment）的缩写，这是一种著名的运行于 Linux、Unix 以及 FreeBSD 等操作系统上的自由图形工作环境，整个系统采用的都是 TrollTech 公司所开发的 Qt 程序库，其具有以下特点：

- 提供了一个美观的现代化桌面。

- 提供了一个具有完整的网络透明性的桌面。
- 提供了一个方便的集成帮助系统，该系统提供了对 KDE 桌面及其应用程序帮助的一致化访问途径。
- 所有的 KDE 应用程序都具有统一的视觉观感。
- 标准化的菜单、工具栏、键盘绑定、颜色样式等。
- 国际化支持，KDE 已拥有 60 余种语言的翻译版本。
- 集中化组织的对话框系统，由具体的桌面配置来运行。
- 大量优秀的 KDE 应用程序。

使用 KDE 作为 X Window 系统的常见 Linux 发行版包括 Kubuntu、Fedora、Mint、openSUSE、Mandriva、Debian 等。2013 年 2 月 6 日，KDE SC 4.10.0 发布。

2. GNOME

GNOME 是另外一种能在 Linux 操作系统上运行的 X Window 应用软件，是 GNU 计划的一部分，其是一种让使用者容易操作和设定电脑环境的工具，目标是基于自由软件，为 Unix 或者类 Unix 操作系统构造一个功能完善、操作简单以及界面友好的桌面环境，是 GNU 计划的正式桌面。GNOME 可以运行在 GNU/Linux（通常叫做 Linux）、Solaris、HP-UX、BSD 和 Apple's Darwin 系统上。GNOME 拥有很多强大的特性，如：高质量的平滑文本渲染，首个国际化和可用性支持，并且包括对反向文本的支持。

采用 GNOME 作为默认 X Window 的 Linux 发行版并不太多，但是其可以在绝大部分发行版上由用户自行安装，图 1.14 是一个使用 GNOME 作为 X Window 应用界面的 Ubuntu 运行界面。



图 1.14 使用 GNOME 的 Ubuntu 操作系统



注 意

通常来说，X Window 只是作为 Linux 操作系统人机交互的一个应用软件，用户可以根据自己的兴趣爱好等实际情况来自行安装对应的 X Window。X Window 和其他应用软件在理论上关系不大，但是在实际应用中由于图形库等依赖关系，常常会导致一些软件在 GNOME 上支持得比 KDE 上好或者反之的情况出现，最典型的例子就是将在第 2 章中介绍的 VIM 和 EMACS。

3. Unity

Unity 在 Ubuntu 中使用，主要被设计成可更高效地使用屏幕空间，与传统的桌面环境相比，消耗的系统资源更少。Unity 将成为 Ubuntu 笔记本版本及新的 Ubuntu Light 即时（instant-on）计算平台的关键组件，其打破了传统的 GNOME 面板配置，其左边包括一个类似 Dock 的启动器和任务管理面板，而面板则由应用程序 Indicator、窗口 Indicator 以及活动窗口的菜单栏组成，如图 1.15 所示。

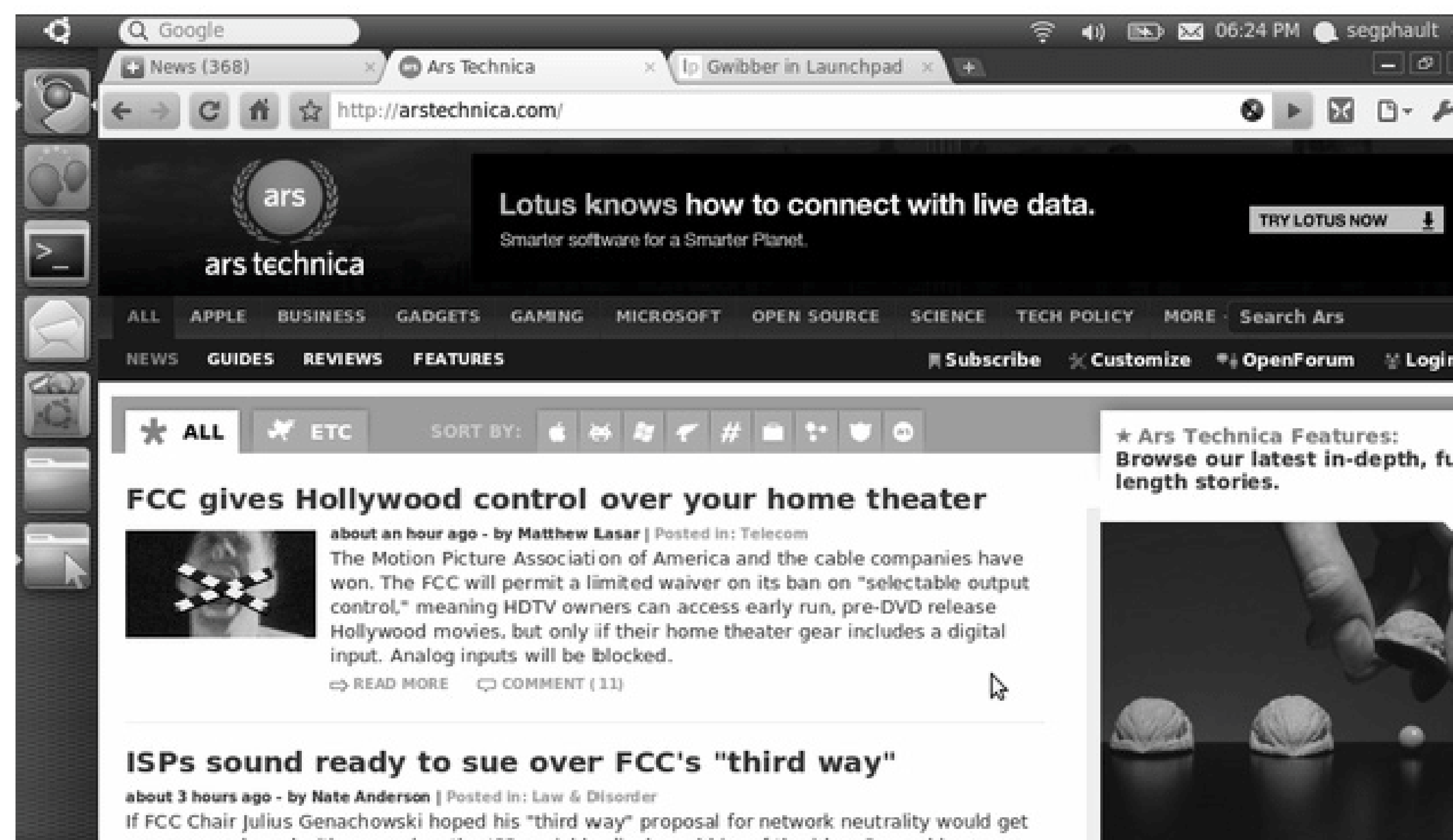


图 1.15　Unity 的运行界面

1.7.2　shell

shell，俗称壳（用来区别于核），是指“提供使用者使用界面”的软件（命令解析器），其类似于 DOS 下的 command.com。它接收用户命令，然后调用相应的应用程序，同时它又是一种程序设计语言。作为命令语言，它交互式地解释和执行用户输入的命令，或者自动解释和执行预先设定好的一连串命令，作为程序设计语言，它定义了各种变量和参数，并提供了许多在高阶语言中才具有的控制结构，包括循环和分支。

shell 并不是 Linux 独有的东西，在 Windows 下也同样存在，shell 不仅仅是以命令行形式出现的，其实 X Windows 也是 shell 的一种，不过在本小节中所特指的 shell 是 Linux 下以命令行形式提供的。

shell 的本质是一个命令解释器，其接收用户命令，然后调用相应的应用程序来执行这些命令，其层次关系如图 1.16 所示。



图 1.16　shell 的层次关系

1. 常见的 shell

在 Linux 的发行版本中最常见的 shell 包括 ash、bash、ksh、csh 和 zsh 共 5 种，它们的关系描述如图 1.17 所示。对图 1.17 的详细说明如下。

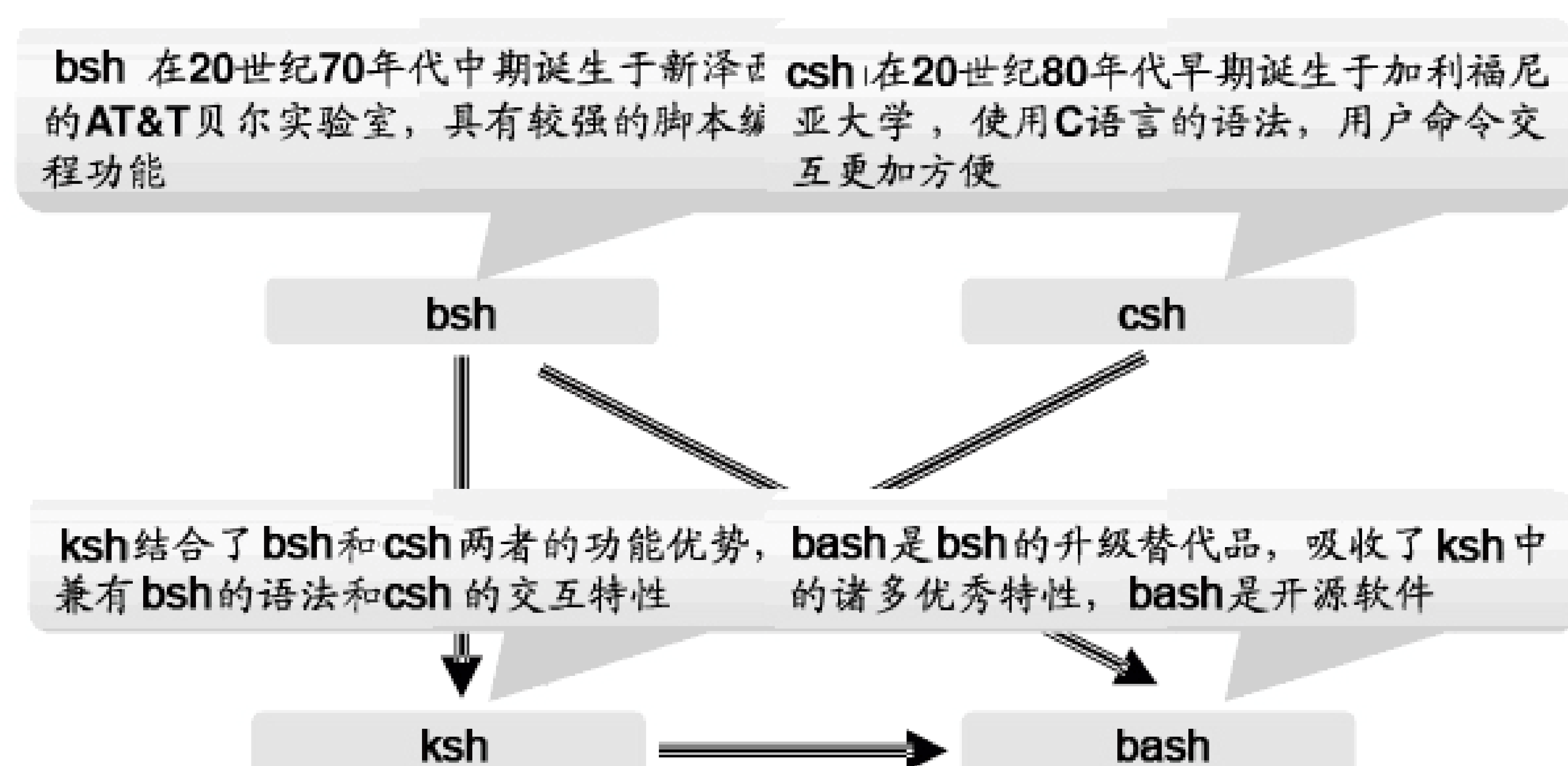


图 1.17　常见 shell 关系说明

- ash: ash shell 是由 Kenneth Almquist 编写的，是 Linux 中占用系统资源最少的一个小 shell，它只包含 24 个内部命令，因而使用起来很不方便。
- bash：bash 是 Linux 系统默认使用的 shell，它由 Brian Fox 和 Chet Ramey 共同完成，是 Bourne Again shell 的缩写，内部命令一共有 40 个。其具有支持上下方向键查阅和快速输入，并修改命令、支持自动查找匹配、可以使用 help 命令调用自带帮助系统的特点。本书所使用的 Ubuntu 12.04 发行版也使用了 bash。
- ksh：是 Korn shell 的缩写，由 Eric Gisin 编写，共有 42 条内部命令。该 shell 最大的优点是几乎和商业发行版的 ksh 完全相容，这样就可以在不用花钱购买商业版本的情况下使用商业版本的性能了。
- csh：其是在 Linux 操作系统中应用比较多的 shell，它由以 William Joy 为代表的共计 47 位作者编写而成，共有 52 个内部命令，该 shell 其实是指向/bin/tcsh 这样的一个 shell，也就是说，csh 其实就是 tcsh。
- zch：这是 Linux 最大的 shell 之一，由 Paul Falstad 完成，共有 84 个内部命令。

2. shell 的启动和使用

shell 在启动的时候，先读取/etc/bash.bashrc 文件对整个 Linux 操作系统进行配置，然后读取$HOME/.bashrc 文件对当前用户进行配置，如果这两个文件有冲突，则以后者为准，这些文件包括以下方面的内容。

- .bash_profile 文件：该文件只被登录用户对应的 shell 所读取，而操作系统内未登录的 shell 只读取.bashrc 文件。
- .bashrc 文件：该文件被启动的所有 shell 所读取。
- .bash_logout 文件：bash 退出时执行该文件。

如果用户安装了多个 shell，则可以在用户管理的相关目录文件中进行设置。和 Linux 内核类似，shell 仅仅只提供了一个计算机和用户进行交互的“内核”，而其具体的命令行输入输出交流要

通过终端来完成。通常来说，在 Linux 操作系统启动的时候，会自动启动多个终端，例如在 Ubuntu 下会同时启动 7 个终端，其中 1~6 号终端均是直接运行的一个“真实终端”，可以使用“Ctrl+Alt+Fn”（Fn = F1~F6）进行切换，而第 7 号终端会交给 X Window 使用，如果想要从 1~6 号终端切换到 X Window 下，只需要使用“Alt + F7”即可，如图 1.18 所示是图形界面下的终端运行界面。

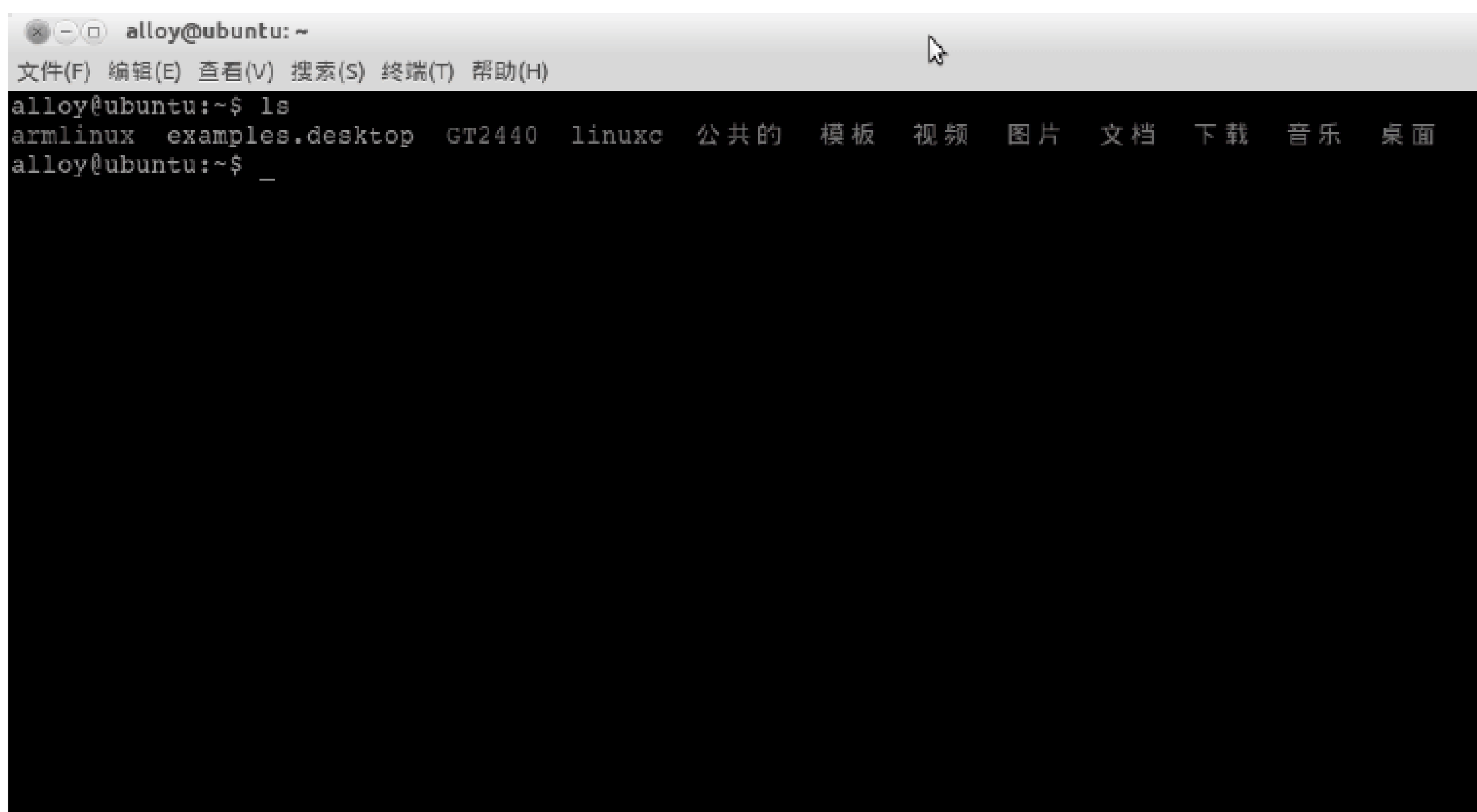


图 1.18　终端的运行界面



注 意

在“真实终端”中通常不能显示中文字符，此时看到的中文字符都是乱码，如果需要显示中文字符，可以使用图形界面下的终端工具。

此外用户还可以使用其他的一些工具通过 Telnet 和 SSH 登录到 Linux 操作系统以完成 shell 下的对应操作，例如可以使用远程登录工具 PuttyMan（菩提曼）。

Telnet 协议是 TCP/IP 协议族中的一员，是 Internet 远程登录服务的标准协议和主要方式。它为用户提供了在本地计算机上完成远程主机工作的能力，在终端使用者的电脑上使用 telnet 程序，用它连接到服务器。终端使用者可以在 telnet 程序中输入命令，这些命令会在服务器上运行，就像直接在服务器的控制台上输入一样。

SSH 则是 Secure Shell 的简称，为建立在应用层和传输层基础上的安全协议，能为用户与 Linux 操作系统的远程连接提供安全可靠的数据传输。

Putty 是 Windows 下非常著名的开源 SSH/Telnet 连接客户端，由于其使用简单，也有不少缺点，例如无法保存密码实现自动登录等，而 PuttyMan 则是其加强版本，增加了添加站点信息、保存密码、多标签、查看和监控操作系统的硬件系统情况等功能，其设置界面和运行界面分别如图 1.19 和图 1.20 所示。

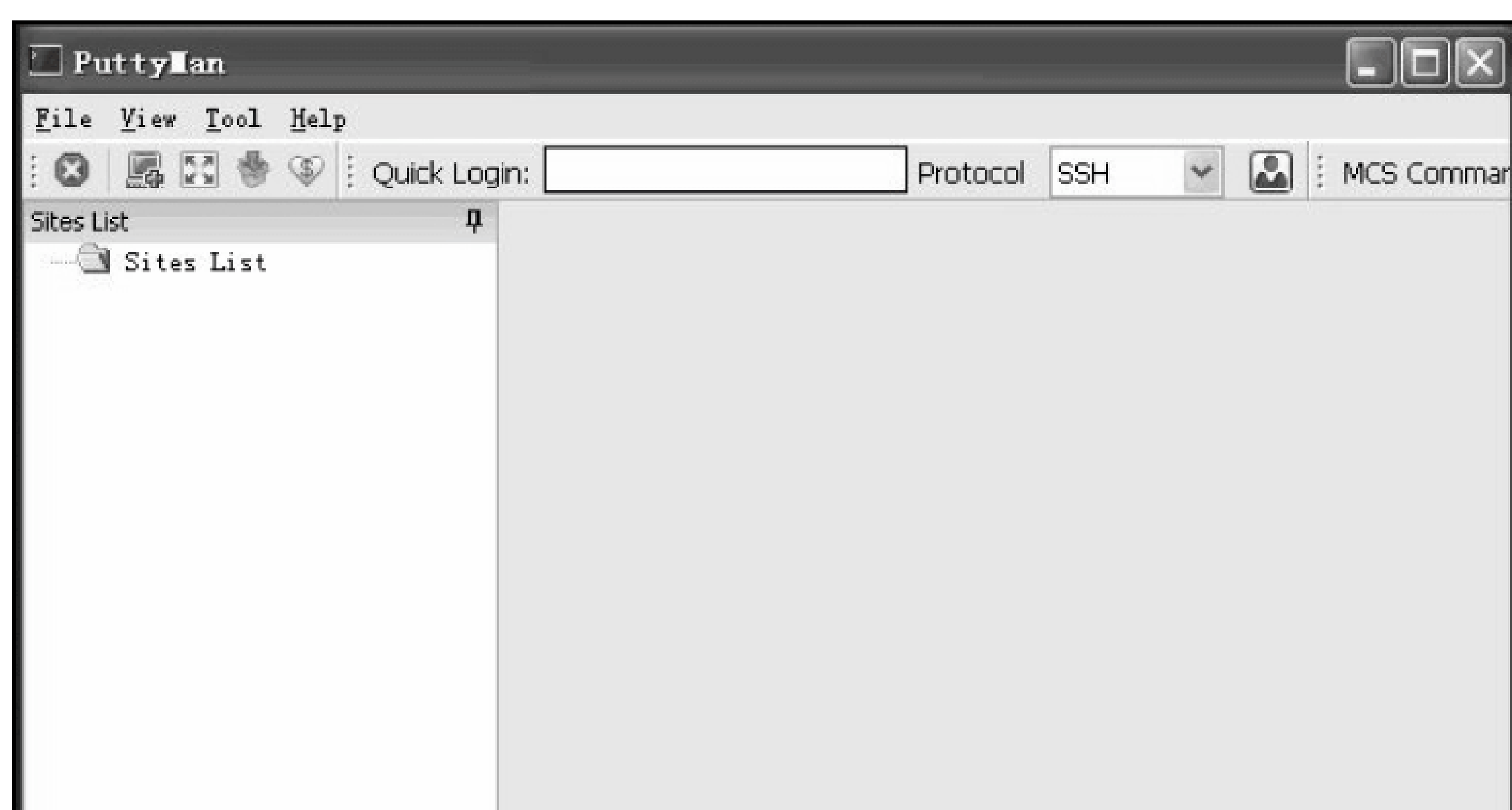


图 1.19　PuttyMan 的设置界面

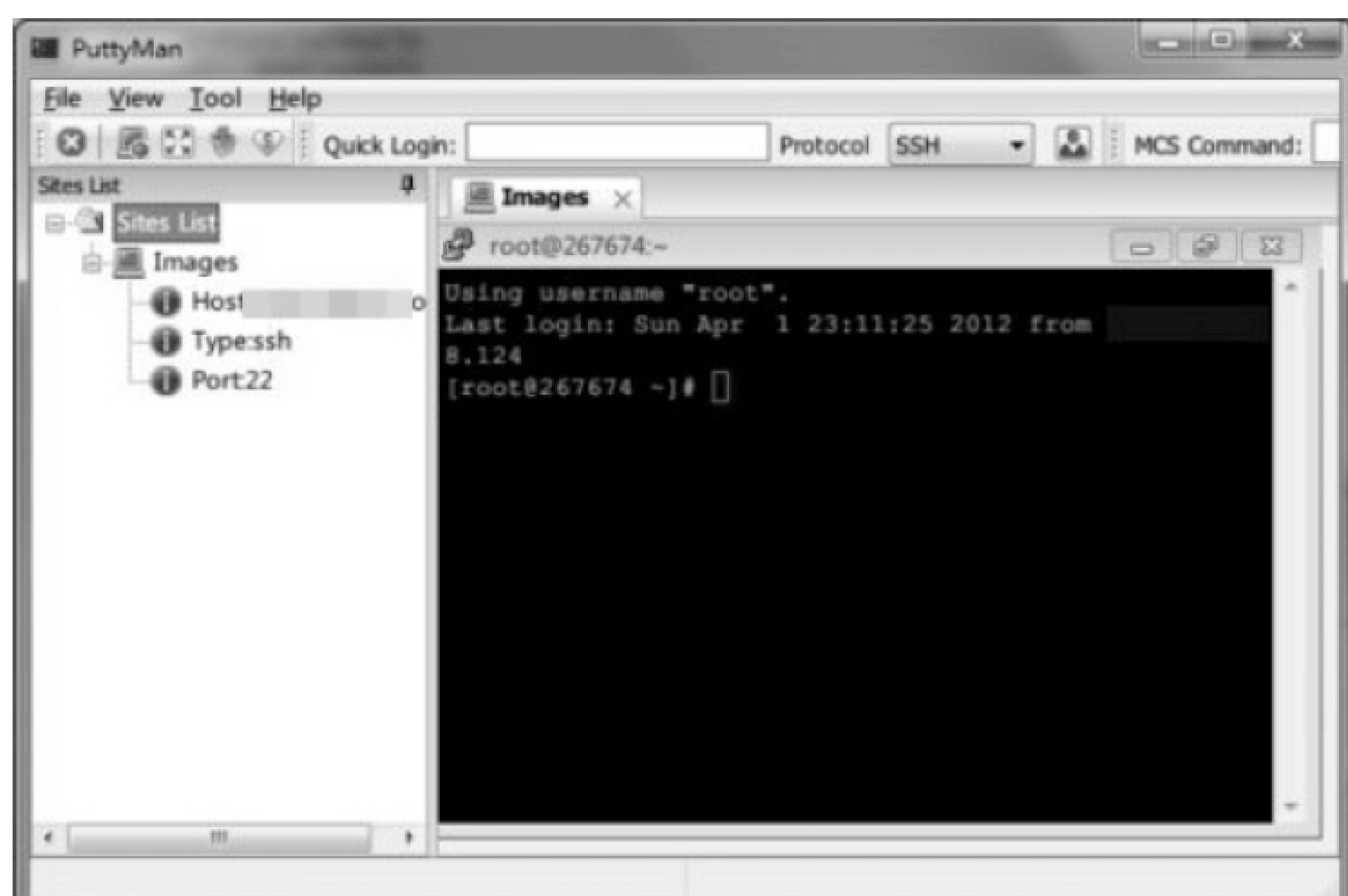


图 1.20　PuttyMan 的运行界面

3. shell 的工作方式

shell 既可以作为命令行提供给用户控制内核完成相应的任务，也可以作为一种编程语言供开发者使用。

- 在命令行工作方式下，shell 识别并且对用户的输入字符串进行响应，以完成相应的工作，这种工作方式通常也被称为“交互式”的工作方式，当用户有输入的时候 shell 才对其做出对应的响应。
- shell 同样可以用作编程语言，在 Linux 中存在一种特殊的可执行文件，其内容是一系列由各种命令组成的纯文本文件（脚本文件），其通常用于完成某些步骤比较多的复杂工作或者是重复性比较强的工作，shell 可以对这些文件进行识别，并且按照设定自动执行相应的动作，这种工作方式通常也被称为“非交互式”的工作方式，不需要用户输入 shell 即可自动做出相应的动作。


注 意

shell 还可以对用户的环境进行配置，这通常会在 shell 的初始化文件中完成，这些配置包括设置窗口属性、快捷键等。

1.8 shell 的使用

熟练使用 shell 是在 Linux 中完成 C 语言开发的基础，本小节将简要介绍 shell 的使用方法，具体的操作命令将在第 1.9 节中进行介绍。

1.8.1 shell 命令的标准格式

shell 和用户交互是以字符串形式存在的命令和命令输出反馈存在的，在 Linux 命令行中输入的第 1 个字必须是一个命令的名字，第 2 个字是命令的选项或参数，命令行中的每个字必须由空格或 Tab 键隔开，格式如下：

```
$ 命令 选项 参数
```

或者：

```
# 命令 选项 参数
```

提示符“$”和“#”区分了用户的不同权限，“$”表示普通用户权限，而“#”代表的是 root 用户（超级用户）权限；选项是包括一个或多个字母的代码，它前面有一个减号（减号是必要的，Linux 用它来区别选项和参数），选项可用于改变命令执行的动作类型。


注 意

在 Ubuntu 操作系统中用户不能直接使用 root 权限，只能通过 sudo 命令来暂时获得 root 权限。

如图 1.21 所示是一个标准的 shell 命令提示符的组成格式。

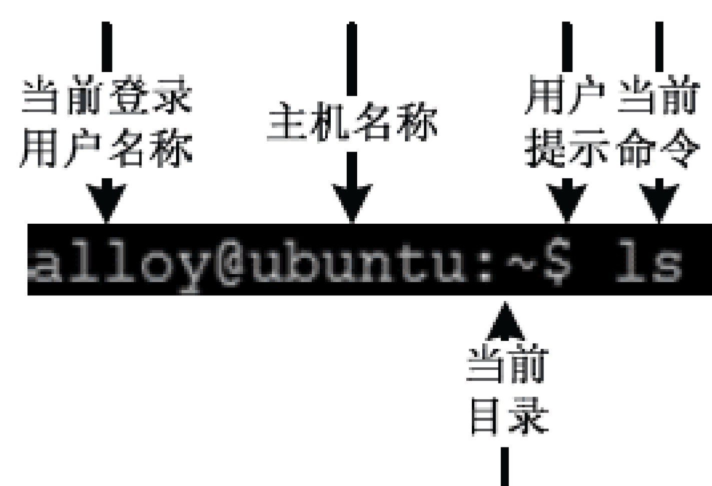


图 1.21 标准 shell 命令提示符的组成

对其各个部分的说明如下。

- alloy：这是当前登录的账户名，会随着登录的具体账户发生改变。
- @：这是一个连接符。
- ubuntu：当前登录到的 Linux 主机名称。
- ~：当前目录，“~”表示的是当前用户的主目录，如果位于其他目录，则会切换到具体

的目录全路径，例如位于当前用户的 linuxc 目录下时命令提示符会变成 alloy@ubuntu:~/linuxc$。

- $：用户提示，普通用户用“$”表示，如果是 root 用户，则用“#”表示。
- ls：一个由当前用户输入的具体的 shell 命令，在用户继续按 Enter 键后执行，这些命令的具体说明请参考第 1.9 章。

命令行实际上是一个可以编辑的文本缓冲区，在按 Enter 键之前，可以对输入的文本进行编辑。例如利用“BackSpace”键可以删除刚键入的字符，也可以进行整行删除，还可以插入字符，使得用户在输入命令（尤其是复杂命令）时，若出现键入错误，不必重新输入整个命令，只要利用编辑操作，即可改正错误。

利用上箭头可以重新显示刚执行的命令，利用这一功能可以重复执行以前执行过的命令，而不必重新键入该命令。

【例 1.1】标准 shell 命令和命令反馈

一个标准的 shell 命令和命令的反馈输出如下，这是利用“ls”命令查看当前文件夹下文件列表的命令反馈输出：

```
alloy@ubuntu:~$ ls
armlinux   examples.desktop   GT2440   linuxc   公共的   模板   视频   图片   文档   下载   音乐   桌面
```

1.8.2 shell 的通配符

在 shell 中除使用普通字符外，还可以使用一些具有特殊含义和功能的字符，称为通配符，在使用它们时应注意其特殊的含义和作用范围。

shell 的通配符主要用于模式匹配，如文件名匹配、路径名搜索、字符串查找等。常用的通配符有“*”、“?”和括在方括号“[ ]”中的字符序列等，用户可以在作为命令参数的文件名中包含这些通配符，构成一个所谓的“模式串”，以便在执行过程中进行模式匹配，这三个通配符的含义分别如下。

- “*”代表任意长度的字符串，例如“L*”匹配以 L 开头的任意字符串。但应注意，文件名中的圆点（.）和路径名中的斜线（/）必须是显式的，即不能用通配符替代它们。例如“*”不能匹配.c，而“.*”才可以匹配.c。
- “?”代表任何单个字符。
- “[ ]”指定了模式串匹配的字符范围，只要文件名中“[ ]”处的字符在指定的范围之内，那么这个文件名就与该模式串匹配。方括号中的字符范围可以由字符串组成，也可以由表示限定范围的起始字符、终止字符及中间连字符（-）组成。例如，f [a-d] 与 f [abcd] 的作用相同。

shell 将把与命令行中指定的模式串相匹配的所有文件名都作为命令参数，形成最终的命令，然后再执行这个命令。如果目录中没有与指定的模式串相匹配的文件名，那么 shell 将使用此模式串本身作为参数传给命令（这正是命令中出现特殊字符的原因所在），表 1.2 列举了这些通配符的具体实例及含义。

表 1.2 通配符的含义

模式串举例	含义
*	当前目录下所有文件的名称
Text	当前目录下所有文件名中含有 Text 字符串的文件名称
[ab-dm]*	当前目录下所有以“a”、“b”、“c”、“d”、“m”开头的文件名称
[ab-dm]?	当前目录下所有以“a”、“b”、“c”、“d”、“m”开头且后面只跟有一个字符的文件名称
/usr/bin/??	目录/usr/bin/下所有名称长度为 2 个字符的文件名称

需要注意的是，中间连字符（-）仅在方括号内有效，表示字符范围，若在方括号外面，就成为普通字符了。而“*”和“?”只在方括号外有效，若出现在方括号之内，它们也失去通配符的能力，成为普通字符了。例如，模式“L[*?]abc”中只有一对方括号是通配符，而“*”和“?”均为普通字符，因此，它匹配的字符串只能是“L*abc”和“L?abc”。

1.8.3 shell 中的引号

在 shell 可以使用的引号包括单引号、双引号和反引号三种。

1. 单引号

由单引号括起来的字符都作为普通字符出现。特殊字符用单引号括起来以后，也会失去原有意义，而只作为普通字符解释。

【例 1.2】单引号的使用

例如在下面的一系列命令中，“$PATH”这个字符串在一般情形下，“$”符号的含义是引用变量的值，PATH 本身是一个 Linux 下的环境变量，其值是一系列的目录，当用户运行某个程序时，Linux 在这些目录下进行搜寻，可以使用 echo 命令来查看变量 PATH 的值：

```
alloy@ubuntu:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

而如果将该字符串放到单引号之中，然后用“=”号赋值给“string”字符串，然后再用 echo 命令来查看“string”字符串的值，可以看到单引号中的字符串保持了其本身的含义，作为普通字符出现。

```
alloy @ubuntu:~/chapter4Exam$ string='$PATH'
alloy@ubuntu:~/chapter4Exam$ echo $string
$PATH
```

2. 双引号

双引号的作用与单引号类似，区别在于它没有那么严格。单引号告诉 shell 忽略所有的特殊字符，而双引号要求忽略大多数字符。具体来说，括在双引号中的三种特殊字符不被忽略：“$”、“\”和“`”，即双引号会解释字符串的特别意义，而单引号则直接使用字符串。如果使用双引号

将字符串赋给变量并反馈它，实际上与直接反馈变量并无差别。如果要查询包含空格的字符串，则经常会用到双引号。

【例 1.3】双引号的使用

首先定义字符变量“x”，然后使用 echo 命令分别显示“$x”、单引号定义的“$x”和双引号定义的“$x”。

```
alloy@ubuntu:~$ x=*          //定义字符变量 x
alloy@ubuntu:~$ echo $x      //显示 x 的值，为当前目录下所有文件夹列表
Desktop Documents Downloads examples.desktop LinuxC Music Pictures Public Templates Videos
alloy@ubuntu:~$ echo '$x'    //使用单引号定义字符变量
$x
alloy@ubuntu:~$ echo "$x"    //使用双引号定义字符变量
*
```

从这个例子中，可以清楚地看出无引号、单引号和双引号之间的区别。

- 第 1 种情况，显示变量 x 的值。由于 x 的值，即字符“*”匹配了当前目录（root 目录）下的所有文件名，故显示变量 x 的值时，即可显示当前目录的所有文件名。
- 第 2 种情况，使用了单引号。单引号中的字符保持其本身的含义，这种情况最简单。
- 最后一种情况，使用了双引号。双引号告诉 shell 在引号内照样进行变量名替换，所以 shell 把“$x”替换为“*”，因为双引号中不做文件名替换（忽略了非特殊字符），所以就把“*”作为要显示的值传递给 echo 命令，作为 echo 命令的参数。

另外，从例子中还可以看到 shell 赋值的先后次序：shell 先进行变量替换，然后进行文件名替换，最后把这些替换值作为参数传递给命令。

3. 反引号

反引号“`”字符所对应的键一般位于键盘的左上角，不要将其同单引号“'”混淆。反引号括起来的字符串被 shell 解释为命令行，在执行时，shell 首先执行该命令行，并以它的标准输出结果取代整个反引号（包括两个反引号）部分。

【例 1.4】反引号的使用

shell 执行 echo 命令时，首先执行“`pwd`”中的命令 pwd，并将输出结果“/”取代“`pwd`”部分，最后输出替换后的整个结果。

```
alloy@ubuntu:/$ pwd
/
alloy@ubuntu:/$ string="current directory is `pwd`"
alloy@ubuntu:/$ echo $string
current directory is /
```

利用反引号的这种功能还可以进行命令置换，即把反引号括起来的执行结果赋值给指定变量。

```
alloy@ubuntu:/$ today=`date`
```

```
alloy@ubuntu:/$ echo today is $today
today is 2012 年  08 月  03 日  星期五  16:58:54 CST
```

另外，反引号还可以嵌套使用。但需要注意的是，嵌套使用时内层的反引号必须用反斜线（\）将其转义。

1.8.4　shell 中的注释符

在 shell 编程或 Linux 的配置文档中，经常要对某些正文行进行注释，以增加程序的可读性。在 shell 中以字符“#”开头的正文行表示注释行。

1.9　Linux 的常用命令

本节将介绍部分 shell 下的常用命令，由于 Linux C 语言程序开发通常是在命令行界面下进行，所以熟练使用这些命令是进行 Linux C 语言程序开发的基础。


注　意

本节内容基于 Ubuntu 12.04，不同的 Linux 发行版可能会略有区别。

1.9.1　目录操作命令

1. 创建目录命令（mkdir）

mkdir 用于在 Linux 系统中建立一个新目录，其标准调用格式如下：

```
mkdir   [选项] 目录
```

mkdir 命令的常用选项如表 1.3 所示。

表 1.3　mkdir 命令常用选项说明

选项	说明
-m	在建立目录时设置目录权限，该目录的权限分为：目录所有者的权限、组中其他人对目录的权限和系统中其他人对目录的权限。这三个权限分别用三个数字之和来表示：对目录的读权限是 4、写权限是 2、执行权限是 1
-p	可以是一个路径名称，此时若路径中的某些目录尚不存在，加上此选项后，系统将自动建立好那些尚不存在的目录，即一次可以建立多个目录

【例 1.5】使用 mkdir 命令建立文件夹

如下所示为使用 mkdir 命令在当前目录(linuxc)中建立一个 chapter1 目录(之前有一个 chapter3 目录）并且将其权限设置为只有文件拥有者才能读写和执行的命令执行过程：

```
alloy@ubuntu:~/linuxc$ ls                          //首先查看当前目录的内容
chapter3                                           //当前只有一个目录 chapter3
alloy@ubuntu:~/linuxc$ mkdir -p -m 700 chapter1    //使用 mkdir 命令创建 chapter1 并且设置权限
```

```
alloy@ubuntu:~/linuxc$ ls -l                          //以详细方式查看当前目录内容
总用量 8
drwx------ 2 alloy alloy 4096   8 月   5 13:13 chapter1
drwxrwxr-x 2 alloy alloy 4096   8 月   5 13:12 chapter3     //可以看到两个目录的权限有区别
```

2. 删除目录命令（rmdir）

与创建目录对应的操作是删除目录，此时可以使用 rmdir 命令，待删除的目录必须为空目录，如果所给的目录不为空，Linux 会报告错误，标准调用格式如下。

```
rmdir [选项] 目录列表
```

rmdir 命令的常用选项如表 1.4 所示。

表 1.4　rmdir 命令常用选项说明

选项	说明
-p	在删除目录表指定的目录后，若父目录为空，则 rmdir 也删除父目录，状态信息将显示哪些被删除，哪些没被删除。



注　意

如果要删除一个非空的目录，应该使用“rm -rf /a/b/c＋ 目录名”命令。

3. 显示当前工作目录命令（pwd）

当前工作目录（路径）对应的是在图形界面中当前打开的文件夹，可以使用 pwd 命令来显示当前工作目录（路径），其标准调用格式如下：

```
pwd
```

【例 1.6】使用 pwd 显示当前工作目录

如下所示为使用 pwd 命令来显示当前工作目录的命令执行过程，可以看到当前的工作目录是/home/alloy/linuxc。

```
alloy@ubuntu:~/linuxc$ pwd          //使用 pwd 命令
/home/alloy/linuxc                  //当前工作目录
```

4. 改变当前工作目录命令（cd）

如果想改变当前工作目录，可以使用 cd 命令，其标准调用格式如下：

```
cd    [directory]
```

其将当前目录改变至 directory 所指定的目录。若没有指定 directory 目录，则回到用户的主目录（例如/home/alloy），需要注意的是为了改变到指定目录，用户必须拥有对指定目录的执行和读权限。该命令可以使用通配符（“*” 和 “？”）。

在 Linux 中有三个特殊目录。

- /：表示 Linux 系统的根目录，是最高级的目录，所有其他目录都是该目录的子目录（或者子目录的子目录），所以在使用 ls 命令查看 usr 文件夹的内容时需要使用“/usr”，因为 usr 是根目录下的一个目录。
- .：表示当前目录。
- ..：表示上一级目录，如果当前目录为/home/alloy，若使用“cd ..”（中间有空格）则会回到/home 目录中。

此外 Linux 中有相对路径和绝对路径两个概念，前者是相对当前目录而言的路径，后者是相对根目录“/”而言的路径。

【例 1.7】使用 cd 切换当前工作目录

如下所示为一个使用 cd 命令在多个目录路径下切换的执行过程：

```
alloy@ubuntu:~/linuxc$ ls                          //首先查看当前目录下的目录
chapter1    chapter3                               //这两个都是目录
alloy@ubuntu:~/linuxc$ cd chapter1                 //切换到 chapter1 目录下
alloy@ubuntu:~/linuxc/chapter1$ cd ../chapter3     //切换到 chapter3 目录下，注意此处使用了相对路径
alloy@ubuntu:~/linuxc/chapter3$ cd ..              //返回上一级目录
alloy@ubuntu:~/linuxc$ cd /                        //切换到根目录
```

1.9.2 文件操作命令

1. 列举文件命令（ls）

ls 用于显示指定工作目录中所包含的文件，其标准调用格式如下：

```
ls [选项]  [文件目录列表]
```

ls 命令的常用选项如表 1.5 所示。

表 1.5 ls 命令常用选项说明

选项	说明
-a	列出目录下的所有文件，包括以“.” 开头的隐含文件
-b	把文件名中不可输出的字符用反斜线加字符编号（就像在 C 语言里一样）的形式列出
-i	输出文件 i 节点的索引信息
-k	以 k 字节的形式表示文件的大小
-l	列出文件的详细信息
-m	横向输出文件名，并以“，”作为分隔符
-n	用数字的 UID、GID 代替名称
-o	显示文件的除组信息外的详细信息
-p -F	在每个文件名后附上一个字符以说明该文件的类型，“*”表示可执行的普通文件；“/”表示目录；“@”表示符号链接；“|”表示 FIFOs；“=”表示套接字（sockets）

（续表）

选项	说明
-q	用“?”代替不可输出的字符
-r	对目录反向排序
-s	在每个文件名后输出该文件的大小
-t	以时间排序
-u	以文件上次被访问的时间排序
-x	按列输出，横向排序
-A	显示除“.”和“..”外的所有文件
-B	不输出以“~”结尾的备份文件
-C	按列输出，纵向排序
-G	输出文件的组信息
-L	列出链接文件名而不是链接到的文件
-N	不限制文件长度
-Q	把输出的文件名用双引号括起来
-R	列出所有子目录下的文件
-S	以文件大小排序
-X	以文件的扩展名（最后一个“.”后的字符）排序
-1	一行只输出一个文件

由于 Linux 支持多种文件类型，每一类用一个字符来表示，其说明如表 1.6 所示。

表 1.6　Linux 的文件类型说明

文件类型	说明
-	常规文件
d	目录
b	块特殊设备
c	字符特殊设备
s	信号灯
m	共享存储器

文件类型的字符表示文件的权限，权限由三个字符串组成，这三个字符串分别表示：该文件所有者的权限、组中其他人的权限和系统中其他人的权限；每个字符串又由三个字符组成，依次表示对文件的读（用字符“r”表示）、写（用字符“W”表示）和执行权限（用字符“x”表示）。当用户没有相应的权限时，该权限的对应位置用短线“-”来表示。 例如：

```
drwxr-x---
```

表示的含义是：“d”表示该文件是目录；目录拥有者的权限是“rwx”（表示有读、写和执行权限）；组中其他人对该目录的权限是“r-x”（表示有读和执行权限，没有写权限），系统中其他人

对该目录的权限是“---”（表示读、写和执行权限都没有）。

【例 1.8】使用 ls 查看目录文件

如下所示为一个使用“ls”命令来显示根目录下文件列表的执行过程，ls 命令后面直接跟上根目录“/”即可：

```
alloy@ubuntu:/$ ls /
bin      dev      initrd.img          lib64          mnt     root   selinux   tmp    vmlinuz
boot     etc      initrd.img.old   lost+found   opt     run     srv          usr    vmlinuz.old
cdrom   home   lib                        media            proc   sbin    sys          var
```

如下所示为使用 ls 加上参数“-l”来使用长格式查看根目录下 usr 子目录中内容的执行过程，在其中可以看到 usr 目录的总用量、各个文件（目录）的权限等：

```
alloy@ubuntu:/$ ls -l usr
总用量 112
drwxr-xr-x     2 root root 36864   8 月   4 18:47 bin
drwxr-xr-x     2 root root    4096   2 月   4 20:00 games
drwxr-xr-x   36 root root    4096   8 月   4 18:47 include
drwxr-xr-x 180 root root 36864   8 月   4 18:29 lib
drwxr-xr-x   10 root root    4096   2 月   4 19:58 local
drwxr-xr-x     2 root root 12288   8 月   4 16:07 sbin
drwxr-xr-x 288 root root 12288   8 月   4 18:47 share
drwxr-xr-x     6 root root    4096   8 月   1 15:32 src
```

2. 查找文件命令（find）

在 Linux 系统中，可以使用 find 命令来查找文件，其标准调用格式如下：

```
find [目录列表]   [匹配标准]
```

find 命令有两个目录列表和匹配标准两个参数，对其说明如下。

- 目录列表：希望查询文件或文件集的目录列表，目录间用空格分隔。
- 匹配标准：希望查询文件的匹配标准或说明，其详细的说明如表 1.7 所示。

表 1.7　find 命令的匹配标准参数说明

选项	说明
-amin n	查找系统中最后 N 分钟访问的文件
-atime n	查找系统中最后 n*24 小时访问的文件
-cmin n	查找系统中最后 N 分钟被改变状态的文件
-ctime n	查找系统中最后 n*24 小时被改变状态的文件
-empty	查找系统中空白的文件，或空白的文件目录，或目录中没有子目录的文件夹
-false	查找系统中总是错误的文件
-fstype type	查找系统中存在于指定文件系统的文件，例如：ext2
-name	使用名称匹配，支持通配符

（续表）

选项	说明
-gid n	查找系统中文件数字组 ID 为 n 的文件
-group gname	查找系统中文件属于 gnam 文件组，并且指定组和 ID 的文件
-daystart	测试系统从今天开始 24 小时以内的文件，用法类似于“-amin”
-depth	使用深度级别的查找过程方式，在某层指定目录中优先查找文件内容
-follow	遵循通配符链接方式查找，另外，也可忽略通配符链接方式查询
-maxdepth levels	在某个层次的目录中按照递减方法查找
-mount	不在文件系统目录中查找

【例 1.9】使用 find 命令查找文件

如下所示为在目录/home/alloy/linuxc/chapter1 下查找 test 文件的命令执行过程：

```
alloy@ubuntu:/$ find /home/alloy/linuxc/chapter1
/home/alloy/linuxc/chapter1
/home/alloy/linuxc/chapter1/test
```

当要查找某个文件时而不知道该文件的全名，只知道这个文件包含几个特定的字母，此时用查找命令也是可找到相应文件的，应该使用通配符，并且使用“-name”匹配标准，如下是使用 find 命令在/dev 目录下查找包含“usb”字符串的文件执行过程。

```
alloy@ubuntu:/$ find /dev -name usb*
/dev/input/by-id/usb-VMware_VMware_Virtual_USB_Mouse-mouse
/dev/input/by-id/usb-VMware_VMware_Virtual_USB_Mouse-event-mouse
/dev/bus/usb
```

3. 显示文件内容命令（cat）

可以使用 cat 命令来显示文件的内容，如果该文件不是文本文件，则可能显示乱码或者出现错误，其标准调用格式如下：

```
cat  [选项]  文件列表
```

cat 命令中的常用选项说明如表 1.8 所示。

表 1.8　cat 命令常用选项说明

选项	说明
-v	利用一种特殊形式显示控制字符，LFD 与 TAB 除外。加了“-v”选项后，“-T”和“-E”选项将起作用。其中：“-T”将 TAB 显示为“ù I”。该选项需要与“-v”选项一起使用，即如果没有使用“-v”选项，则这个选项将被忽略。“-E”在每行的末尾显示一个 $ 符，该选项需要与“-v”选项一起使用
-n	在文件的每行前面显示行号

【例 1.10】使用 cat 命令查看文件内容

如下所示为使用 cat 命令来显示 C 语言文件 exam301Open.c 的操作过程，使用“-n”参数在每一行之前加上了行编号。

```
alloy@ubuntu:~$ cat linuxc/chapter3/exam301Open.c -n
1    //这是一个标准的 open 函数调用实例，打开文件 opentest，如果没有则创建
2    //返回文件的描述符，并且关闭文件后退出
3    #include <stdlib.h>
4    #include <fcntl.h>
5    #include <stdio.h>
6    int main(void)
7    {
   ……  //此处省略部分输出
13       exit(0);                //退出
14   }
```

cat 命令还可以用于将两个文件连接到一起放到另外一个文件中去，如下所示为使用 cat 命令把 exam301Open.c 文件和文本文件 test.txt 拼接后将其内容存放到一个新的文本文件 cattest.txt 中，然后查看 cattest.txt 文件内容的操作过程。

```
alloy@ubuntu:~/linuxc/chapter3$ cat test.txt    //查看 test.txt 文件内容
this is a test!
this is a test!
alloy@ubuntu:~/linuxc/chapter3$ cat exam301Open.c test.txt > cattest.txt   //拼接，使用“>”写入新文件
alloy@ubuntu:~/linuxc/chapter3$ cat cattest.txt -n   //查看合并之后的文件内容
 1   //这是一个标准的 open 函数调用实例，打开文件 opentest，如果没有则创建
 2   //返回文件的描述符，并且关闭文件后退出
 3   #include <stdlib.h>
 4   #include <fcntl.h>
 5   #include <stdio.h>
 6   int main(void)
……  //此处省略部分输出
14   }
15   this is a test!                        //从这里开始是第 2 个文件的内容
16   this is a test!
```

4. 复制文件命令（cp）

Linux 下的 cp 命令用于复制文件或目录，其可以把指定的源文件复制到目标文件或把多个源文件复制到目标目录中，其标准调用格式如下：

```
cp [选项] 源文件或目录 目标文件或目录
```

cp 命令的常用选项说明如表 1.9 所示。

表 1.9 cp 命令中的常用选项说明

选项	说明
-a	该选项通常在复制目录时使用，它保留链接、文件属性，并递归地复制目录，其作用等于 dpR 选项的组合
-d	复制时保留链接
-f	删除已经存在的目标文件而不提示
-i	和 f 选项相反，在覆盖目标文件之前将给出提示要求用户确认，回答 y 时目标文件将被覆盖，是交互式拷贝
-p	此时 cp 除复制源文件的内容外，还将把其修改时间和访问权限也复制到新文件中
-r	若给出的源文件是一个目录文件，此时 cp 将递归复制该目录下所有的子目录和文件，此时目标文件必须为一个目录名
-l	不进行复制，只是链接文件


注 意

为防止用户在不经意的情况下利用 cp 命令破坏另一个文件，如果指定的目标文件名是一个已存在的文件名，利用 cp 命令复制文件后，这个文件就会被新复制的源文件覆盖，因此，在使用 cp 命令复制文件时，最好使用“-i”选项。

5. 移动和重命名文件命令（mv）

可以使用 mv 命令来移动文件，还可以同时修改文件名称，即把源文件以一个新文件名移动到另一个新的目录中去，其标准调用格式如下：

```
mv [选项] 源文件名 目标文件名
mv [选项] 源目录名 目标目录名 2
mv [选项] 文件列表 目录
```

mv 命令的选项说明如表 1.10 所示。

表 1.10 mv 命令选项说明

选项	说明
-b	当遇到要覆盖其他文件或目录时，将自动备份，备份文件名为原文件名加上“-S”参数指定的字符串，若未设置则加上“~”
-i	交互模式，当移动的目录已存在同名的目标文件名时，利用覆盖的方式写文件，但在写入之前给出提示
-f	通常情况下，目标文件存在但用户没有写权限时，mv 会给出提示。本选项会使用 mv 命令执行移动而不给出提示
-u	当要覆盖的文件或目录比源文件要新，则不覆盖目标文件
-S <字符串>	指定备份文件名后要加上的字符串

6. 文件内容统计命令（wc）

wc 命令可以统计指定文件中的字节数、字数、行数，并将统计结果显示输出，其标准调用格式如下：

```
wc  [选项] 文件列表
```

该命令用于统计给定文件中的字节数、字数、行数。如果没有给出文件名，则从标准输入（通常是键盘）读取，wc 同时也给出所有指定文件的总统计数。字是由空格字符区分开的最大字符串，对 wc 命令的选项说明如表 1.11 所示。

表 1.11　wc 命令选项说明

选项	说明
-c	统计字节数
-l	统计行数
-w	统计字数

【例 1.11】使用 wc 命令统计文件内容

使用 wc 命令对./linuxc/chapter3 目录下的 C 语言文件 exam301Open.c 和可执行文件 exam301Open 进行统计的执行过程如下，对于 C 语言文件分别统计了其中的字节数、行数和字数。

```
alloy@ubuntu:~$ wc -c ./linuxc/chapter3/exam301Open.c
555 ./linuxc/chapter3/exam301Open.c
alloy@ubuntu:~$ wc -l ./linuxc/chapter3/exam301Open.c
14 ./linuxc/chapter3/exam301Open.c
alloy@ubuntu:~$ wc -w ./linuxc/chapter3/exam301Open.c
34 ./linuxc/chapter3/exam301Open.c
alloy@ubuntu:~$ wc ./linuxc/chapter3/exam301Open.c
  14   34 555 ./linuxc/chapter3/exam301Open.c
alloy@ubuntu:~$ wc ./linuxc/chapter3/exam301Open
     9    63 8536 ./linuxc/chapter3/exam301Open
```

7. 删除文件命令（rm）

如果要删除一个文件，可以使用 rm 命令，其标准调用格式如下：

```
rm  [选项] 文件
```

该命令用于删除一个指定的文件(通常来说并不删除目录)，其常用的选项说明如表 1.12 所示。

表 1.12　rm 命令选项说明

选项	说明
-f	强制删除一个文件
-i	删除文件之前进行提示


注　意

如果想要删除一个非空的目录（文件夹），可以使用如下的命令串："rm -rf /a/b/c＋ 文件夹"。

1.9.3　其他命令

1. 用户切换命令（su 和 sudo）

Linux 是一种多用户操作系统，如果所有用户共享一个账号，会造成许多麻烦，因此在 Linux 中每个用户都有自己的账号，各个用户的账号可以根据需要分配不同的权限。Linux 提供了与之相关的用户操作命令。su 命令可以用来切换用户身份，其标准调用格式如下：

```
su  [选项]  user
```

除 root 外，其他用户切换身份时，都需要输入密码，su 命令的常用选项说明如表 1.13 所示。

表 1.13　su 命令常用选项说明

选项	说明
-p	执行 su 时不改变环境参数
-c	切换到 user 用户并执行指令（command），然后再切换回原来的用户
-s	指定要执行的 shell，默认在/etc/passwd 文件中已设置完成，若用户需要更改 Shell 时，可采用此参数

sudo 命令用来以系统管理员的身份执行指令，其标准调用格式如下：

```
sudo [选项]  命令
```

以系统管理者的身份执行指令，也就是说，经由 sudo 所执行的指令就好像是 root 亲自执行，sudo 命令的常用选项说明如表 1.14 所示。

表 1.14　sudo 命令选项说明

选项	说明
-l	显示出执行 sudo 的用户权限
-v	sudo 在第一次执行时或是在 N 分钟内没有执行（N 预设为 5）会问密码，这个参数是需要重新进行一次确认，如果超过 N 分钟，也会询问密码
-k	强迫用户在下一次执行 sudo 时询问密码（不论有没有超过 N 分钟）


注　意

Ubuntu 锁定了 root 用户，所以不能使用 su 命令切换用户，只能使用 sudo 命令来临时获得 root 权限。

2. 进程管理命令（ps 和 kill）

ps 命令用于显示当前系统中由该用户运行的进程列表，而 kill 命令用于输出特定的信号给指定进程号（PID）的进程，并根据该信号完成指定的行为，其中可能的信号有进程挂起、进程等待、

进程终止等，它们的标准调用格式如下：

```
ps：ps [选项]
kill：kill [选项]   进程号(PID)
```

ps 命令的参数说明如表 1.15 所示。

表 1.15 ps 命令选项说明

选项	说明
-ef	查看所有进程及其 PID（进程号）、系统时间、命令的详细目录、执行者等
-aux	除可显示“-ef”所有内容外，还可显示 CPU 及内存占用率、进程状态
-w	以加宽方式显示，这样可以显示较多的信息

kill 的常用选项参数如表 1.16 所示，其命令中的进程号为信号输出的指定进程号，当选项缺省时输出终止信号给该进程。

表 1.16 kill 命令的常用选项说明

选项	说明
-s	将指定信号发送给进程

【例 1.12】ps 和 kill 命令的使用

在命令行中，输入以下命令（需要注意的是 ps 和 kill 命令都可能需要 root 权限，此时也需要加上 sudo 才能执行），系统将会显示所有的进程（以下显示不完整）：

```
alloy@ubuntu:~$ ps –ef
UID        PID    PPID    C    STIME    TTY     TIME        CMD
root         1       0    0    2005     ?       00:00:05    init
root         2       1    0    2005     ?       00:00:00    [keventd]
root         3       0    0    2005     ?       00:00:00    [ksoftirqd_CPU0]
root         4       0    0    2005     ?       00:00:00    [ksoftirqd_CPU1]
root      7421       1    0    2005     ?       00:00:00    /usr/local/bin/ntpd -c /etc/ntp
root     21787   21739    0    17:16    pts/1   00:00:00    grep ntp
```

接下来可以终止进程号为 7421 的 ntp 进程，输入如下命令：

```
alloy@ubuntu:~$ kill 7421
```

之后再次查看，使用命令如下：

```
alloy@ubuntu:~$ ps –ef | grep ntp
```

系统输出：

```
root     21789 21739    0 17:16 pts/1      00:00:00 grep ntp
```

可以看出，已经没有该进程号的进程，说明该进程已经被删除。


注 意

ps 命令在使用中通常可以与其他一些命令结合起来，主要作用是提高效率。ps 选项中的参数 w 可以写多次，通常最多写 3 次，它的含义表示加宽 3 次，这足以显示很长的命令行了，例如："ps －auxwww"。

3. IP 地址管理命令（ifconfig）

ifconfig 命令用于查看和配置网络接口的地址和参数，包括 IP 地址、网络掩码、广播地址，它的使用权限是超级用户，其有两种使用格式，分别用于查看和更改网络接口，标准调用格式如下。

- ifconfig [选项] [网络接口]：用来查看当前系统的网络配置情况。
- ifconfig 网络接口 [选项] 地址：用来配置指定接口（如 eth0、eth1）的 IP 地址、网络掩码、广播地址等。

ifconfig 的第 2 种使用方式的常见选项说明如表 1.17 所示。

表 1.17　ifconfig 命令的常见选项说明

选项	说明
interface	指定的网络接口名，如 eth0 和 eth1
up	激活指定的网络接口卡
down	关闭指定的网络接口卡
broadcast address	设置接口的广播地址
point to point	启用点对点方式
address	设置指定接口设备的 IP 地址
netmask address	设置接口的子网掩码地址

【例 1.13】使用 ifconfig 命令查看当前系统的网络配置

如下所示为使用 ifconfig 查看当前系统网络配置的操作过程，第一个 eth0 中是有线网卡的相关网络信息，可以看到其 IP 地址为 192.168.0.111，MAC 地址（网卡硬件地址）为 00:0c:29:4e:97:50；第二个 lo 为 Linux 的自身环回地址，固定为 127.0.0.1，访问这个地址即可访问自己。

```
alloy@ubuntu:~$ ifconfig
eth0      Link encap:以太网  硬件地址 00:0c:29:4e:97:50
          inet 地址:192.168.0.111  广播:192.168.0.255  掩码:255.255.255.0
          inet6 地址: fe80::20c:29ff:fe4e:9750/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:1525 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:1365 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:173902 (173.9 KB)  发送字节:186990 (186.9 KB)

lo        Link encap:本地环回
          inet 地址:127.0.0.1  掩码:255.0.0.0
          inet6 地址: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  跃点数:1
```

```
    接收数据包:52 错误:0 丢弃:0 过载:0 帧数:0
    发送数据包:52 错误:0 丢弃:0 过载:0 载波:0
    碰撞:0 发送队列长度:0
    接收字节:3654 (3.6 KB)  发送字节:3654 (3.6 KB)
```

4. 帮助命令（man）

对于绝大部分 Linux 终端用户和 C 语言程序员而言，经常需要查询一些命令或者函数的具体使用方法，此时可以使用 Linux 自带的 man 帮助文件命令。

只要在命令 man 后，输入想要获取的命令名称（例如 ls），man 就会列出一份完整的说明，其内容包括命令语法、各选项的意义以及相关命令等，其标准调用格式如下：

```
man    [选项]   命令名称
```

man 命令的常用选项说明如表 1.18 所示。

表 1.18　man 命令的常用选项说明

选项	说明
-f	只显示出命令的功能，而不显示其中详细的说明文件
-w	不显示手册页，只显示对应帮助文件的所在位置
-a	显示所有的手册页，而不是只显示第一个
-E	在每行的末尾显示“$”符号

【例 1.14】使用 man 命令查看 ls 命令的使用方法

以下是使用 man 命令外加“-f”和“-w”选项查看 ls 命令的执行过程，使用“-f”选项的时候只显示了 ls 的功能，使用“-w”选项则显示了 ls 命令的对应帮助文件所在位置（/usr/share/man/man1/ls.1.gz）。

```
alloy@ubuntu:~$ man -f ls
ls (1)                    - list directory contents
alloy@ubuntu:~$ man -w ls
/usr/share/man/man1/ls.1.gz
```

5. 关机和重启命令（shutdown、halt 和 reboot）

由于 Linux 是一种多用户、多任务操作系统，因此在切断计算机电源之前，必须先关闭 Linux 系统。决不能不执行关机进程就切断计算机电源，这样做会导致保存在内存缓冲区中的磁盘数据来不及写回磁盘，从而破坏文件系统。本节将介绍一下与关机和重启计算机有关的命令。

shutdown 命令可以安全地关闭或重启 Linux 系统，它在系统关闭之前给系统上的所有登录用户发送一条警告信息。该命令还允许用户指定一个时间参数，可以是一个精确的时间，也可以是从现在开始的一个时间段。精确时间的格式是 hh:mm，表示小时和分钟；时间段由“+”和分钟数表示。系统执行该命令后，会自动进行数据同步的工作，其标准调用格式如下：

```
shutdown [选项] [时间] [警告信息]
```

shutdown 命令的常用选项说明如表 1.19 所示。

表 1.19 shutdown 命令常用选项说明

选项	说明
-k	并不真正关机，而只是发出警告信息给所有用户
-r	关机后立即重新启动
-h	关机后不重新启动
-c	取消一个已经运行的 shutdown



关机命令需要 root 权限。

halt 是最简单的关机命令，其实际上是调用“shutdown –h”命令。halt 执行时，“杀死”应用进程，文件系统写操作完成后就会停止内核，其标准调用格式如下：

```
halt [选项]
```

halt 命令的常用选项说明如表 1.20 所示。

表 1.20 halt 命令选项说明

选项	说明
-n	在关机前不做将内存资料写回硬盘的动作
-w	并不会真的关机，只是把记录写到 /var/log/wtmp 文件里
-d	不把记录写到 /var/log/wtmp 档案里（参数“-n”包含了“-d”）
-f	强迫关机，不调用 shutdown 这个指令
-i	在关机之前先把所有网络相关的装置停止
-p	当关机的时候，顺便做关闭电源（poweroff）的动作，取消一个已经运行的 shutdown



halt 命令同样需要超级用户权限。

reboot 命令用来重新启动计算机，其标准调用格式如下：

```
reboot [选项]
```

reboot 命令的常用选项说明如表 1.21 所示。

表 1.21 reboot 命令常用选项说明

选项	说明
-n	在关机前不做将内存资料写回硬盘的动作
-w	并不会真的关机，只是把记录写到 /var/log/wtmp 文件里
-d	不把记录写到 /var/log/wtmp 档案里（参数“-n”包含了“-d”）
-f	强迫关机，不调用 shutdown 指令
-i	在关机之前先把所有网络相关的装置停止

6. 查看内核和发行版版本号命令（uname 和 lsb_release）

可以使用 uname 来查看系统的相关信息，其相关选项的参数说明如表 1.22 所示。

表 1.22　uname 命令的选项说明

选项	说明
-a	显示全部
-s	显示内核名称
-n	显示网络节点主机名称
-r	显示内核发行版
-v	显示内核版本号
-m	显示系统硬件主机名称
-p	显示处理器名称

【例 1.15】使用 uname 命令查看当前系统信息

以下是使用 uname 命令查看当前系统信息的执行过程，首先分别使用“-s”、“-n”等选项参数来显示当前系统信息的对应分项内容。

```
alloy@ubuntu:~$ uname -s     //显示系统内核名称
Linux
alloy@ubuntu:~$ uname -n     //显示网络节点主机名称
ubuntu
alloy@ubuntu:~$ uname -r     //显示内核发行版本号
3.11.0-26-generic
alloy@ubuntu:~$ uname -v     //显示内核版本号
#45~precise1-Ubuntu SMP Tue Jul 15 04:02:35 UTC 2014
alloy@ubuntu:~$ uname -m     //显示系统硬件主机名称
x86_64
alloy@ubuntu:~$ uname -p     //显示处理器类型和型号(由于此处使用的是虚拟机,所以只能显示 x86_64)
x86_64
alloy@ubuntu:~$ uname -a     //显示全部，可以看到是之前的分项内容集合
Linux ubuntu 3.11.0-26-generic #45~precise1-Ubuntu SMP Tue Jul 15 04:02:35
UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
alloy@ubuntu:~$
```

除了 uname 命令之外，还可以使用 lsb_release 命令来查看操作系统对应的发行版信息，其相关参数如表 1.23 所示，需要注意的是这个命令需要 root 权限。

表 1.23　lsb_release 命令选项说明

选项	说明
-v	显示版本号
-i	显示发行版作者
-d	显示当前使用版本的相应描述
-r	显示当前使用版本的发行版本号
-a	显示全部

【例 1.16】使用 lsb_release 命令查看操作系统信息

以下是使用 lsb_release 命令来查看操作系统相应信息的执行过程，分别使用“-i”、“-d”等选

项参数来显示当前操作系统信息对应的分项内容，其中第一次使用 lsb_release 命令时需要输入密码。

```
alloy@ubuntu:~$ sudo lsb_release -i      //显示发行版作者
[sudo] password for alloy:                //要求输入密码
Distributor ID: Ubuntu
alloy@ubuntu:~$ sudo lsb_release -d      //显示当前使用版本的相应描述
Description:      Ubuntu 12.04.4 LTS
alloy@ubuntu:~$ sudo lsb_release -r       //显示发行版本号
Release:           12.04
alloy@ubuntu:~$ sudo lsb_release -v       //显示版本号，此时不能获得 LSB 模块
No LSB modules are available.
alloy@ubuntu:~$ sudo lsb_release -a       //显示之前的各个分项内容集合
No LSB modules are available.
Distributor ID: Ubuntu
Description:      Ubuntu 12.04.4 LTS
Release:           12.04
Codename:           precise
```

1.10 本章习题

1. 请解释如下术语：GNU、GPL 和 POSIX。
2. 请描述 Linux 的组成。
3. 请列举三个以上的常见 Linux 发行版本，并简要说明它们的特点。
4. 请列举最常用的两种 Linux 图形界面，并简要说明它们的特点。
5. 请解释如下术语：shell、SSH 和 Putty。
6. 请解释 shell 命令中三种引号的区别。
7. 在 Linux 的终端界面下，使用 mkdir 命令在 Linux 的用户目录下建立一个名称为 Temp 的目录，使用 cd 命令切换到该目录下，然后用 rmdir 命令删除该目录。
8. 在 Linux 的终端界面下，使用 ls 命令列举根目录下所有文件和文件夹的详细信息。
9. 在 Linux 的终端界面下，使用 man 命令查看 touch 命令的详细使用方法。
10. 在 Linux 的终端界面下，使用 shutdown 命令关闭 Linux 系统。

第 2 章　在 Linux 下进行 C 语言开发

C 语言最早是由贝尔实验室的 Dennis Ritchie 为了 Unix 的辅助开发而编写的，它是在 B 语言的基础上开发出来的，尽管C语言不是专门针对 Unix 操作系统或机器编写的，但它与 Unix 系统的关系十分紧密。由于它的硬件无关性和可移植性，使 C 语言逐渐成为世界上使用最广泛的计算机语言。本章将介绍 Linux 下 C 语言开发的基础知识，涉及的内容包括：

- C 语言的特点以及开发流程。
- Linux 下的 C 语言编辑、编译、调试以及项目管理工具。
- Linux 下 C 语言程序的运行机制、内存管理和分配、输入输出、系统调用和库函数等。

2.1　C 语言的特点和开发流程

为了进一步规范 C 语言的硬件无关性，1987 年美国国家标准协会（ANSI）根据 C 语言问世以来各种版本对 C 语言的发展和扩充，制定了新的标准，称为 ANSI C。ANSI C 语言比原来的标准 C 语言有了很大的发展。目前流行的 C 语言编译系统都是以它为基础的。

C 语言的成功并不是偶然的，它强大的功能和它的可移植性让它能在各种硬件平台上游刃有余，总体而言，C 语言具有结合了高级语言的基本结构和低级语言的使用性、结构化、数据类型多、功能齐全以及可移植性强等特点。

在 Linux 中开发一个 C 语言应用程序的流程如图 2.1 所示，其中每个环节的详细说明如下。

- 需求分析，算法设计：先根据应用代码需要实现的功能进行需求分析，并且根据需求设计出相应的算法。
- 程序代码编辑：在文本编辑器中输入 C 程序源代码并保存。
- 编译：把源程序编译成目标程序，并且检查其中的语法错误，如果其中有语法错误，则需要返回修改程序代码，然后再次编译。
- 功能逻辑调试：语法没有错并不代表程序代码没有错误，此时的代码并不一定能实现预先设定的功能，必须进行相应的功能逻辑测试以确定达到了预定的目标，此时可能会借助一些调试工具或者调试手段；如果没能达到预期的目标，则需要返回程序代码编辑修改代码。
- 链接，生成可执行文件：在确定代码编写已经没有问题之后，需要通过链接生成对应的可执行文件。

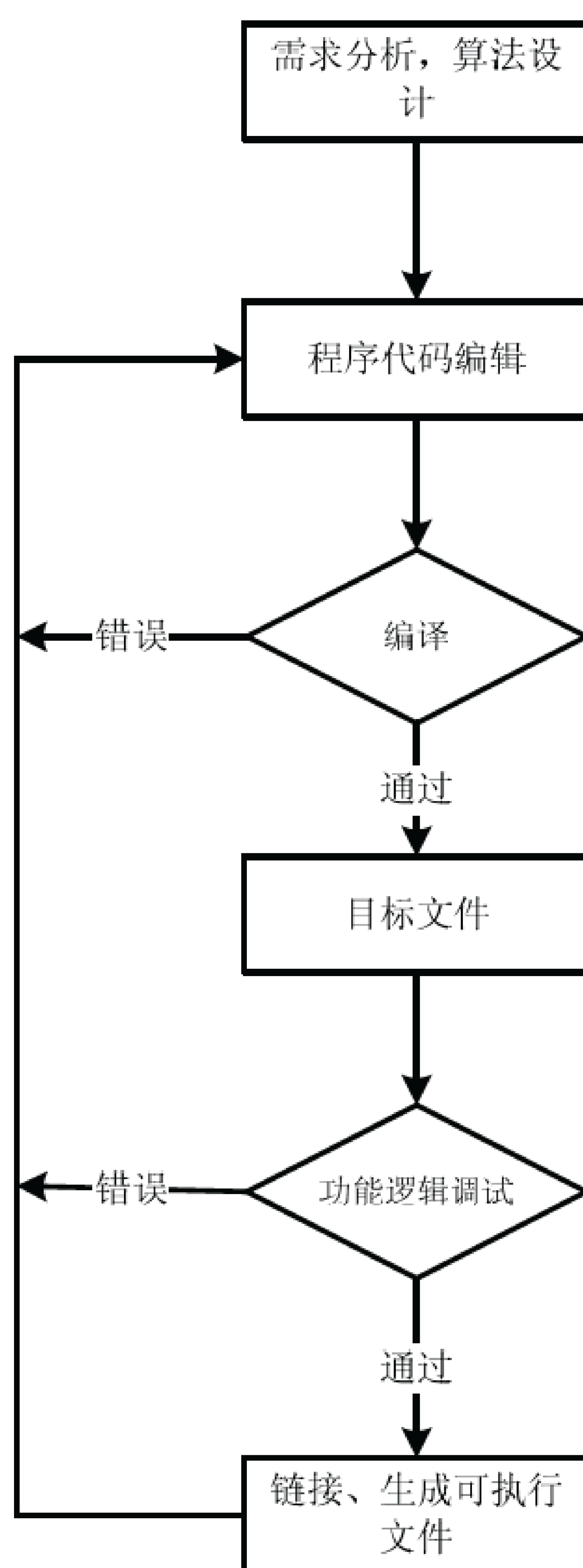


图 2.1　Linux 中的 C 语言开发流程

2.2　Linux 下的 C 语言开发工具

Linux 为软件开发者提供了强大的 C 语言开发环境和丰富的开发维护工具，熟悉并掌握这些工具是进行 Linux 平台软件开发的必要条件，这些工具包括：

- 编辑工具：Linux 系统提供了许多文本编辑程序，可以完成对代码的录入编辑工作，比较常用的有 vim 和 emacs 等，本章的第 2.3 节将对这些编辑工具的基础使用方法进行介绍，这里不再赘述。
- 编译工具：编译是指将 C 语言的源代码转换为可执行代码的过程，其涉及的工作和文件如图 2.2 所示，包括了词法/语法和语义的分析、中间代码的生成和优化、符号表的管理和出错处理等。在 Linux 中，最常用的编译器是 gcc 编译器。它是 GNU 推出的功能强大、性能优越的多平台编译器，其执行效率与一般的编译器相比平均效率要高 20%～30%，第 2.4 节将对 gcc 编译工具的使用方法进行简单介绍，这里不再赘述。

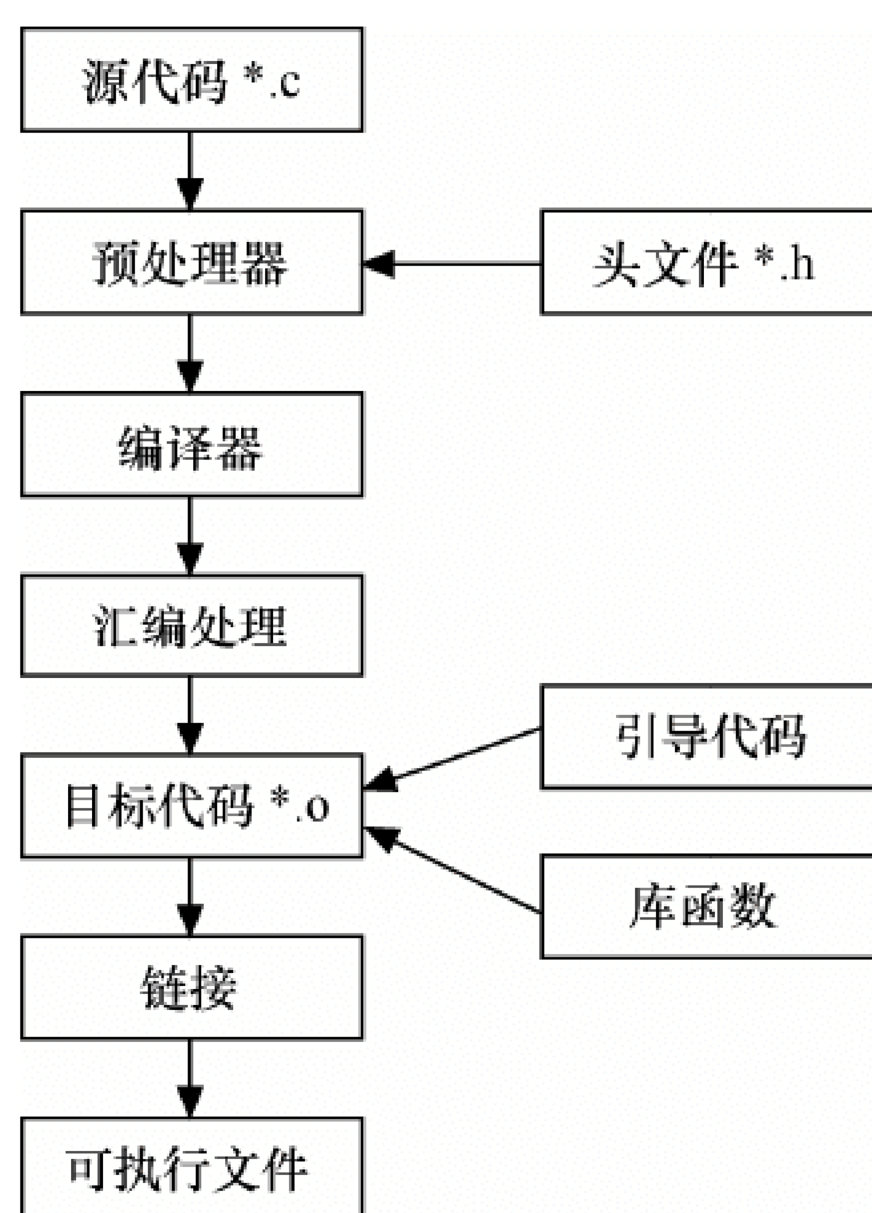


图 2.2　编译工作的功能

- 调试工具：调试工具用于方便程序员对 C 语言的目标代码进行调试，在编程的过程中，调试所消耗的时间往往远远大于编写代码的时间，因此一个功能强大、使用方便的调试器是必不可少的，GDB 是绝大多数 Linux 开发人员所使用的调试器，它可以方便地设置断点、单步跟踪等，第 2.5 节将对 GDB 调试工具的使用方法进行简单介绍，这里不再赘述。
- 项目管理和维护工具：make 等是一种控制编译或者重复编译软件的工具，此外还能自动管理软件编译的内容、方式和时机，其可以对 C 语言的程序源文件进行有效的管理，熟练使用这些工具可以大大减少开发者的工作量，本书将在第 12 章中对其进行详细介绍，这里不再赘述。

在 Linux 的桌面环境下通常还提供了 C 语言的集成开发环境（IDE），这是一种集编辑工具、编译工具、调试工具和项目管理维护工具于一体的大型应用软件，如果开发者在 Windows 系统中做过软件开发，则一定不会对它感到陌生。在 Linux 下可以用于 C 语言程序开发的常见 IDE 包括 CodeBlocks、CodeLite、Anjuta、Eclipse 等，其中 CodeBlocks、CodeLite 与 Windows 系统中的 Visual Studio 界面非常类似，比较容易上手。

2.3　Linux C 语言的代码编辑工具

在 Linux 中开发 C 语言应用代码时，首先需要进行源代码的编写，此时需要一个代码编辑器，其实质就是一个文本编辑器，只不过增加了一些代码编辑的辅助功能，例如关键字高亮、补齐等。在 Linux 中最常见的代码编辑器包括 vim、emacs 等，本节将简单介绍它们的基础使用方法。

2.3.1　vim

vim 是“Vi IMproved”的简称，其是 vi 编辑器的加强版，提供了执行输入、输出、删除、查找、替换、块操作等众多文本操作，用户还可以根据自己的需要对其进行定制。



本书采用了 vim 作为代码编辑器，其是 Unix/Linux 下最基本的文本编辑器，工作在字符模式下，由于不需要图形界面，使它成为效率很高的文本编辑器。尽管在 Linux 上也有很多图形界面的编辑器可用，但 vim 在系统和服务器管理应用中的功能是那些图形编辑器无法比拟的，所以在本书中也仅仅对 vim 的基础使用方法进行了较为详细的介绍，而对其他代码编辑器仅进行概述。

1. vim 的启动和退出

在 Linux 终端命令提示符下输入 vim（或 vim+文件名），即可启动 vim 编辑器，例如：

```
vim filename
```

或者：

```
vim
```

按下回车键后，Linux 便会自动打开文件名为“filename”的文件的 vim 编辑界面，其启动界面如图 2.3 所示。

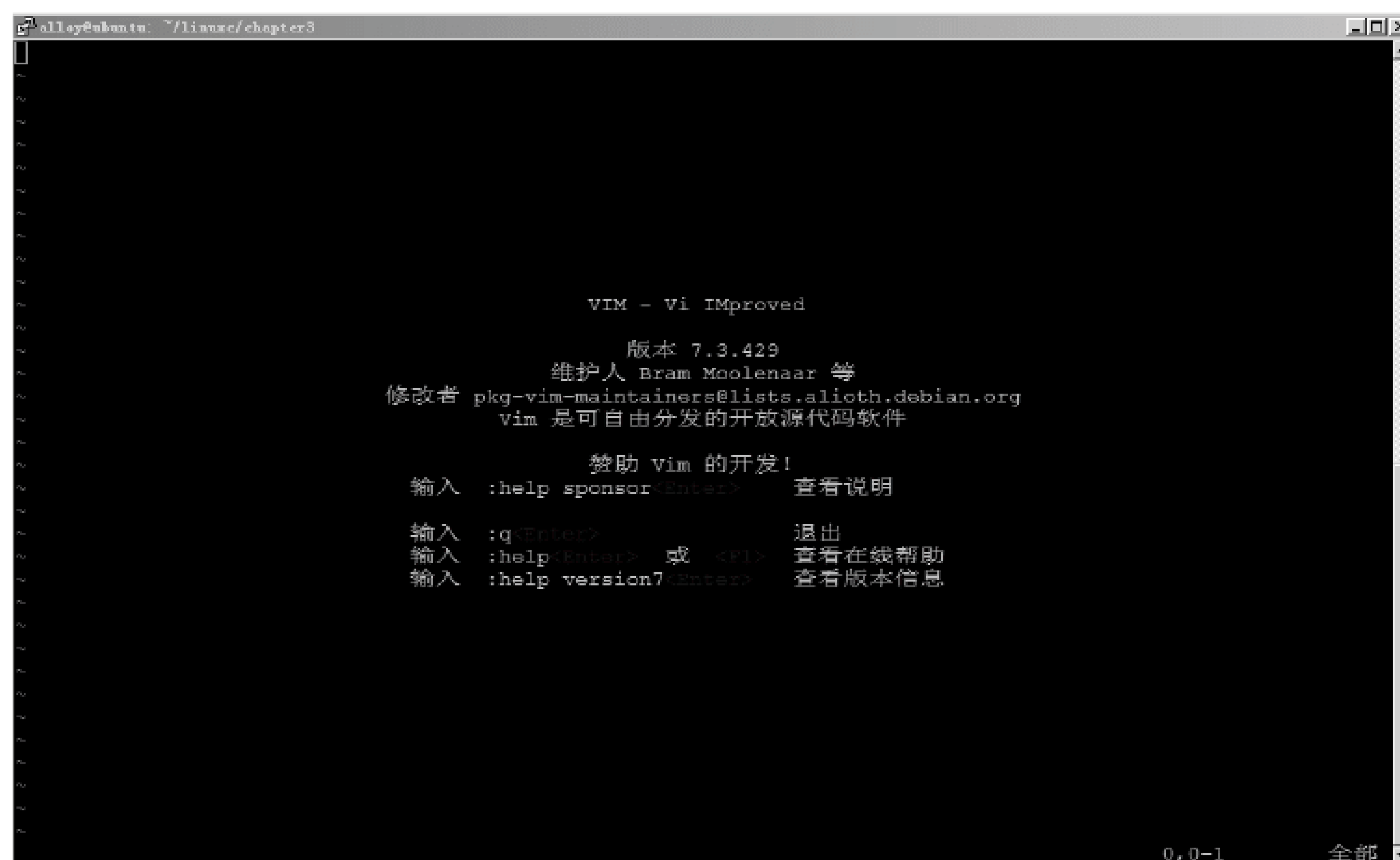


图 2.3　vim 的启动界面

当使用“vim+文件名”的命令来启动 vim 时，若进行编辑的是当前工作目录下已存在的文件，启动后即可看到该文件中的内容；若是当前目录下不存在的文件，则系统首先创建该文件，再使用 vim 进行编辑。

若要退出 vim，必须先按下“Esc”键回到 vim 的命令行工作模式（关于 vim 的工作模式请参考下一小节），然后键入“:”，此时光标会停留在最下面一行（底行模式），再键入“q”，最后按下“Enter”键即可退出 vim。

2. vim 的工作模式及其切换

vim 拥有三种工作模式：命令行工作模式（command mode）、插入工作模式（input mode）与

底行工作模式（last line mode），对这三种工作模式下的功能描述如下。

- 命令行工作模式：也称为“普通模式”，启动 vim 后默认进入此模式，在该模式下可以使用隐式命令（命令不显示）来实现光标的移动、复制、粘贴、删除等操作，但在该模式下，编辑器并不接受用户从键盘输入的任何字符来作为文档的编辑内容，也就是说并不能将 C 语言代码输入到文件。
- 插入工作模式：在该工作模式下，用户输入的任何字符都被认为是编辑到某一个文件的内容，并直接显示在 vim 的文本编辑区，在该模式下可以将 C 语言代码输入到文件。
- 底行工作模式：在该工作模式下，用户输入的任何字符串都会被当成命令，会在 vim 的最下面一行显示，按下回车键后便会执行该命令，如果该字符串并不是一个有效的命令，则会出现错误提示。

使用 vim 编辑器，首先必须能够熟练掌握各种工作模式的用途以及各种工作模式间的切换，如图 2.4 所示为 vim 在三种工作模式间的切换方法。

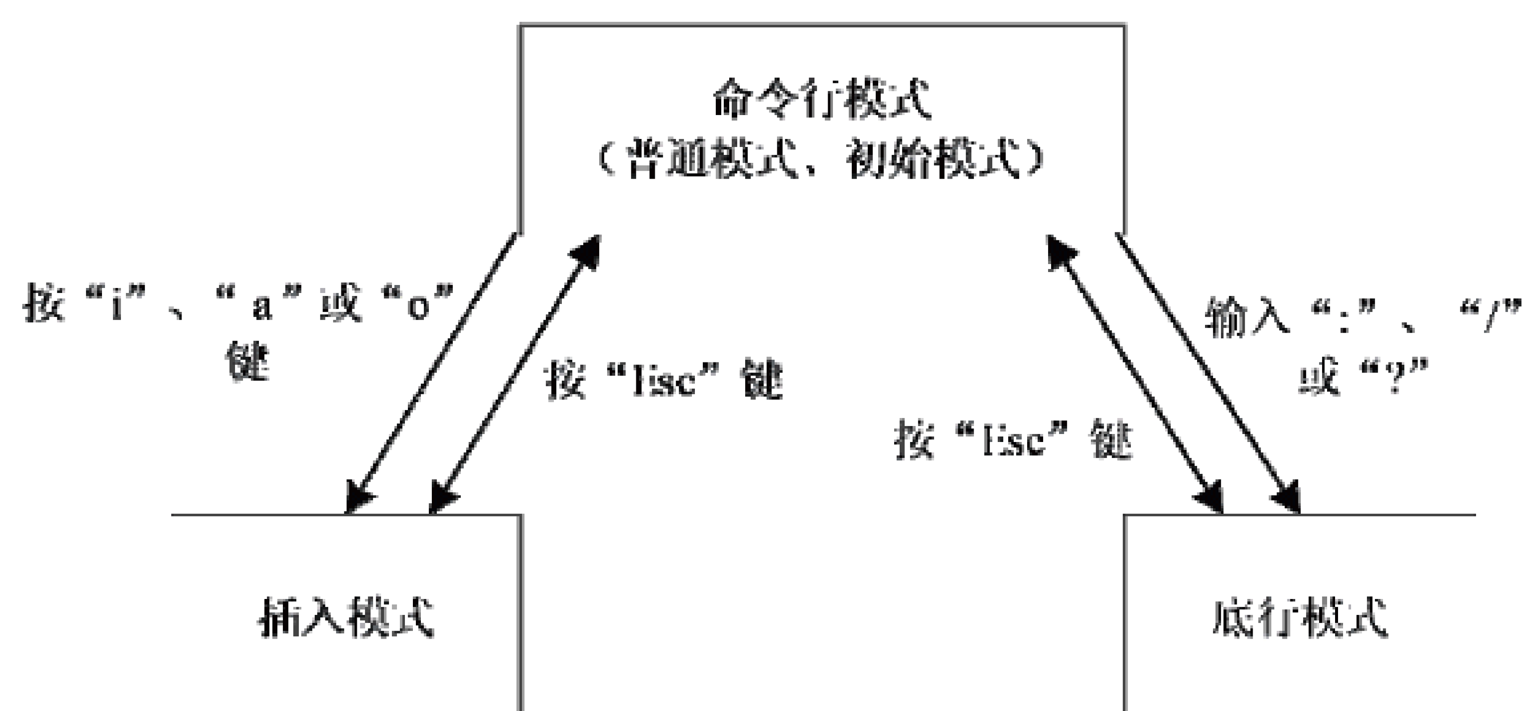


图 2.4　vim 在三种工作模式间的切换方法

从图 2.4 中可以看到，命令行工作模式是 vim 编辑器的初始模式，从该模式可以实现到任何模式的切换；而插入模式和底行模式之间不能相互切换，因为在插入模式下，任何输入的字符都被认为是编辑到某一个文件的内容，而不是命令；在底行模式下，任何输入的字符都被看作是底行命令（尽管可能是不合法的），二者都必须先通过命令行模式才能进入对方，即需要先按下“Esc”键回到初始模式。

3. vim 的命令行工作模式

vim 在命令行工作模式下的主要操作是使用方向键或快捷键对当前光标进行定位，以及使用相应的命令对当前文件中的文本进行诸如复制、删除、粘贴等基础编辑操作，这些命令说明如表 2.1~表 2.4 所示。


注 意

命令行工作模式下的命令比较多，在此仅做简单介绍，用户在使用时也可以查阅帮助文档。

在命令行工作模式下，可以通过使用上、下、左、右共 4 个方向键来移动光标的位置，但是在类似使用 telnet 远程登录等场合下就没办法使用方向键，此时必须用命令行模式下的光标移动命令，这些命令对应的字符串和操作说明如表 2.1 所示。

表 2.1　移动光标的常用命令

命令	操作说明
h	向左移动光标
l	向右移动光标
j	向下移动光标
k	向上移动光标
^	将光标移动到该行的开头（第一个非空字符）
$	将光标移动到该行行尾，同键盘上的“End”键
0	将光标移动到该行行首，同键盘上的“Home”键
G	将光标移动到文档最后一行的开头（第一个非空字符）
nG	将光标移动到文档的第 n 行的开头（第一个非空字符），n 为正整数
w	光标向后移动一个字（单词）
nw	光标向后移动 n 个字（单词），n 为正整数
b	光标向前移动一个字（单词）
nb	光标向前移动 n 个字（单词），n 为正整数
e	将光标移动到本单词的最后一个字符。如果光标所在的位置为本单词的最后一个字符，则跳到下一个单词的最后一个字符，“.”、“,”、“#”、“/”等特殊字符都会被当成一个字
{	光标移动到前面的“{”处，这在使用 vim 进行 C 语言编程时很适用
}	同“{”的使用，将光标移动到后面的“}” 处
Ctrl+b	向上翻一页，相当于 Page Up
Ctrl+f	向下翻一页，相当于 Page Down
Ctrl+u	向上移动半页
Ctrl+d	向下移动半页
Ctrl+e	向下翻一行
Ctrl+y	向上翻一行

复制、粘贴是在编辑文档时最常用的操作之一，可以大大节约用户重复输入的时间。对 vim 的命令行工作模式下常用的复制、粘贴命令对应的字符串和操作说明如表 2.2 所示。

表 2.2　复制、粘贴的常用命令

命令	操作说明
yy	复制光标所在行的整行内容
yw	复制光标所在单词的内容
nyy	复制从光标所在行开始向下的 n 行内容，n 为正整数，表示复制的行数
nyw	复制从光标所在字开始向后的 n 个字，n 为正整数，表示复制的字数
p	粘贴，将复制的内容粘贴在光标所在的位置

在 vim 编辑器中，可以一次删除一个字符，也可以一次删除多个字符和整行，在 vim 命令行工作模式下常用的删除命令对应的字符串和操作说明如表 2.3 所示。

表 2.3　删除文本的常用命令

命令	操作说明
x	删除光标所在位置的字符，同键盘上的“Delete”键
X	删除光标所在位置的前一个字符
nx	删除光标所在位置及其后的 n-1 个字符，n 为正整数
nX	删除光标所在位置及其前的 n-1 个字符，n 为正整数
dw	删除光标所在位置的单词
ndw	删除光标所在位置及其后的 n-1 个单词，n 为正整数
d0	删除当前行光标所在位置前的所有字符
d$	删除当前行光标所在位置后的所有字符
dd	删除光标所在行
ndd	删除光标所在行及其向下的 n-1 行，n 为正整数
nd+上方向键	删除光标所在行及其向上的 n 行，n 为正整数
nd+下方向键	删除光标所在行及其向下的 n 行，n 为正整数

vim 在命令行工作模式下还提供了一些其他常用的命令，包括字符替换、撤销操作、符号匹配等，其对应的字符串和操作说明如表 2.4 所示。

表 2.4　其他常用命令

命令	操作说明
r	替换光标所在位置的字符，例如 rx 是指将光标所在位置的字符替换为 x
R	替换光标所到之处的字符，直到按下“Esc”键为止
u	表示恢复功能，即撤销上一次的操作
U	取消对当前行所做的所有改变
.	重复执行上一次的命令
ZZ	保存文档后退出 vim 编辑器
%	符号匹配功能，在编辑时若输入“%(”，则系统会自动匹配相应的“)”

4. vim 的插入工作模式

在插入工作模式下 vim 没有复杂的命令，用户从键盘输入的任何有效字符都被看作是写进当前正在编辑的文件中的内容，并显示在 vim 的文本编辑区，也就是说，只有在插入模式下，才可以进行文字的输入操作。如表 2.5 所示为从命令行模式切换至插入模式的几个常用命令，在插入工作模式下，随时可以使用“Esc”键回到 vim 的命令行工作模式。

表 2.5 命令行工作模式切换至插入工作模式的命令

命令	操作说明
i	从光标所在的位置开始插入新的字符
I	从光标所在行的行首开始插入新的字符
a	从光标所在位置的下一个字符开始插入新的输入字符
A	从光标所在行的行尾开始插入新的字符
o	增加一行，并将光标移动到下一行的开头开始插入字符
O	在当前行的上面增加一行，并将光标移动到上一行的开头开始插入字符

5. vim 的底行工作模式

vim 的底行工作模式也被称为“最后行模式”，是指可以在界面最底部的一行输入控制操作命令，主要用来进行一些文字编辑的辅助功能，例如字符串搜索、替代、保存文件，以及退出 vim 等。

在命令行工作模式下输入冒号“:”，或者是使用“？ ”和“/”键，即可进入底行工作模式，底行工作模式下的常用命令对应的字符串和操作说明如表 2.6 所示。

表 2.6 底行工作模式下的常用命令

命令	操作说明
q	退出 vim 程序，如果文件有过修改，则必须先保存文件
q!	强制退出 vim 而不保存文件
x	保存文件并退出 vim（exit）
x!	强制保存文件并退出 vim
w	保存文件，但不退出 vim（write）
w!	对于只读文件，强制保存修改的内容，但不退出 vim
wq	保存文件并退出 vim，同 x
E	在 vim 中创建新的文件，并为文件命名
N	在本 vim 窗口中打开新的文件
w filename	另存为 filename 文件，不退出 vim
w! filename	强制另存为 filename 文件，不退出 vim
r filename	读入 filename 指定的文件内容插入到光标位置（read）
set nu	在 vim 的每行开头处显示行号
s/pattern1/pattern2/g	将光标当前行的字符串 pattern1 替换为 pattern2
%s/pattern1/pattern2/g	将所有行的字符串 pattern1 替换为 pattern2
g/parttern1/s//parttern2	将所有行的字符串 pattern1 替换为 pattern2
num1,num2 s/pattern1/pattern2/g	将从行 num1~num2 的字符串 partten1 替换为 partten2
/	查找匹配字符串功能，利用“/字符串”的命令模式，系统便会自动查找，并突出显示所有找到的字符串，然后转到找到的第一个字符串。如果想继续向下查找，可以按 n 键；向前继续查找则按 N 键
?	也可以使用“？ 字符串”查找特定字符串，它的使用与“/ 字符串”相似，但它是向前查找字符串

6. vim 的操作步骤

使用 vim 来编辑一个 C 语言源代码文件时的基础应用操作步骤如下：

01 使用“vim+文件名”命令启动 vim，并且创建/打开一个 C 语言文件，此时 vim 位于命令行工作模式。

02 使用“a”命令进入 vim 的插入工作模式。

03 在插入工作模式下对 C 语言源文件的内容进行编辑。

04 使用“Esc”键退出 vim 的插入工作模式，进入底行工作模式。

05 在底行工作模式下使用“:+wq”命令保存并且退出 vim。

7. 使用 vim 的配置文件

当打开一个 C 语言源文件（.C）的时候，用户通常希望其能提供一些关键字的提示，例如把“#include”等利用特殊的颜色表示出来，还希望其能将对应的大括号配对等，而刚刚安装的 vim 编辑环境也许并不具备这方面的功能，此时可以通过对 vim 的配置文件进行编辑，以达到该目的。

vim 的配置文件是一个 vimrc 文件，启动 vim，进入命令行模式，输入“echo $VIM”命令可以查看到当前 vim 编辑的配置文件所在位置，通常来说位于/usr/share/vim 路径下，如图 2.5 所示。

```
~
/usr/share/vim
```

图 2.5　vim 的配置文件位置

使用 ls 命令查看/usr/share/vim 路径下的文件，可以看到 vimrc 文件，在该路径下还存放了包括 vim 可执行文件在内的一系列其他文件。

```
alloy@ubuntu:~$ ls /usr/share/vim/
addons   registry   vim73   vimcurrent   vimfiles   vimrc   vimrc.tiny
```

使用 vim 打开 vimrc 文件，需要注意的是要修改该文件后保存，则需要使用 sudo 命令以获得 root 权限。

```
alloy@ubuntu:~$ sudo vim /usr/share/vim/vimrc
```

可以看到 vimrc 文件内容如下，其前面进行了一些对 vim 的描述，然后使用“"”（引号）作为注释符使得一些配置方法没有生效（如果去掉“"”则会使这些配置生效），我们在这个文件中使用汉字进行了相应的介绍：

```
" All system-wide defaults are set in $VIMRUNTIME/debian.vim (usually just
" /usr/share/vim/vimcurrent/debian.vim) and sourced by the call to :runtime
" you can find below. If you wish to change any of those settings, you should
" do it in this file (/etc/vim/vimrc), since debian.vim will be overwritten
" everytime an upgrade of the vim packages is performed. It is recommended to
" make changes after sourcing debian.vim since it alters the value of the
" 'compatible' option.
——以上是一些关于 vim 的介绍，可以不予理会
" This line should not be removed as it ensures that various options are
" properly set to work with the Vim-related packages available in Debian.
```

```
runtime! debian.vim
——说明本 vim 是 debian 发行版
" Uncomment the next line to make Vim more Vi-compatible
" NOTE: debian.vim sets 'nocompatible'. Setting 'compatible' changes numerous
" options, so any other options should be set AFTER setting 'compatible'.
"set compatible
——前面 3 行用于说明和 vi 的兼容性，第 4 行是用于设置和 vi 编辑器兼容性的选项
" Vim5 and later versions support syntax highlighting. Uncommenting the next
" line enables syntax highlighting by default.
"syntax on
——前面 2 行用于说明 vim5 版本之后开始支持类型高亮显示，最后 1 行用于设置类型高亮显示
" If using a dark background within the editing area and syntax highlighting
" turn on this option as well
"set background=dark
——前面 2 行用于说明背景色的设置，最后 1 行用于设置背景色
" Uncomment the following to have Vim jump to the last position when
" reopening a file
"if has("autocmd")
"   au BufReadPost * if line("'\"") > 1 && line("'\"") <= line("$") | exe "normal! g'\"" | endif
"endif
——第 1 行用于说明 vim 对跳转到上一次位置的支持，后面 4 行用于设置跳转
" Uncomment the following to have Vim load indentation rules and plugins
" according to the detected filetype.
"if has("autocmd")
"   filetype plugin indent on
"endif
——前 2 行用于设置 vim 打开文件的规则，后面 3 行用于设置打开文件
" The following are commented out as they cause vim to behave a lot
" differently from regular Vi. They are highly recommended though.
"set showcmd                    " Show (partial) command in status line.
"set showmatch                  " Show matching brackets.
"set ignorecase                 " Do case insensitive matching
"set smartcase                  " Do smart case matching
"set incsearch                  " Incremental search
"set autowrite                  " Automatically save before commands like :next and :make
"set hidden                     " Hide buffers when they are abandoned
"set mouse=a                    " Enable mouse usage (all modes)
——前 2 行用于说明后面的设置，后面是关于 vim 的一些常规设置，下面我们会详细介绍
" Source a global configuration file if available
if filereadable("/etc/vim/vimrc.local")
  source /etc/vim/vimrc.local
endif
——关于配置文件位置的说明
```

为了达到方便编写 C 语言源文件的目的，应该对 vimrc 文件中的如下选项进行设置（去掉该选项前的“"”即可）：

- "syntax on：打开文件类型高亮显示，打开之后会对 C 语言的关键字使用特殊颜色显示。
- "set showmatch：显示配对的括号。

- "set nu: 显示行号，这句是 vimrc 文件中没有的，需要自己添加。
- "set autoindent: 打开换行自动缩进。
- "set cindent: 按照 C 语言的书写习惯自动缩进，这个缩进是按照 8 个空格来进行的。
- "set mouse = a: 打开鼠标支持，此时在 vim 中使用滚轮和点击均可。

完成以上操作之后保存 vimrc 文件，再次打开一个 C 语言源文件，是不是看到了我们想要的改变呢？



网上有许多其他用户已做好的、功能更加复杂的 vim 配置文件下载，有兴趣的读者可以去自行研究。

2.3.2 Emacs

Emacs，即 Editor Macros（编辑器宏）的缩写，是 Linux 下一个功能强大的图形化文本编辑器软件，可以用来编写 C 语言源程序。与 vim 相比，其显著特点是可以使用鼠标进行大部分的操作，对于习惯使用 Windows 系统的用户来说，Emacs 是一个不错的选择，图 2.6 是 Emacs 的运行界面。

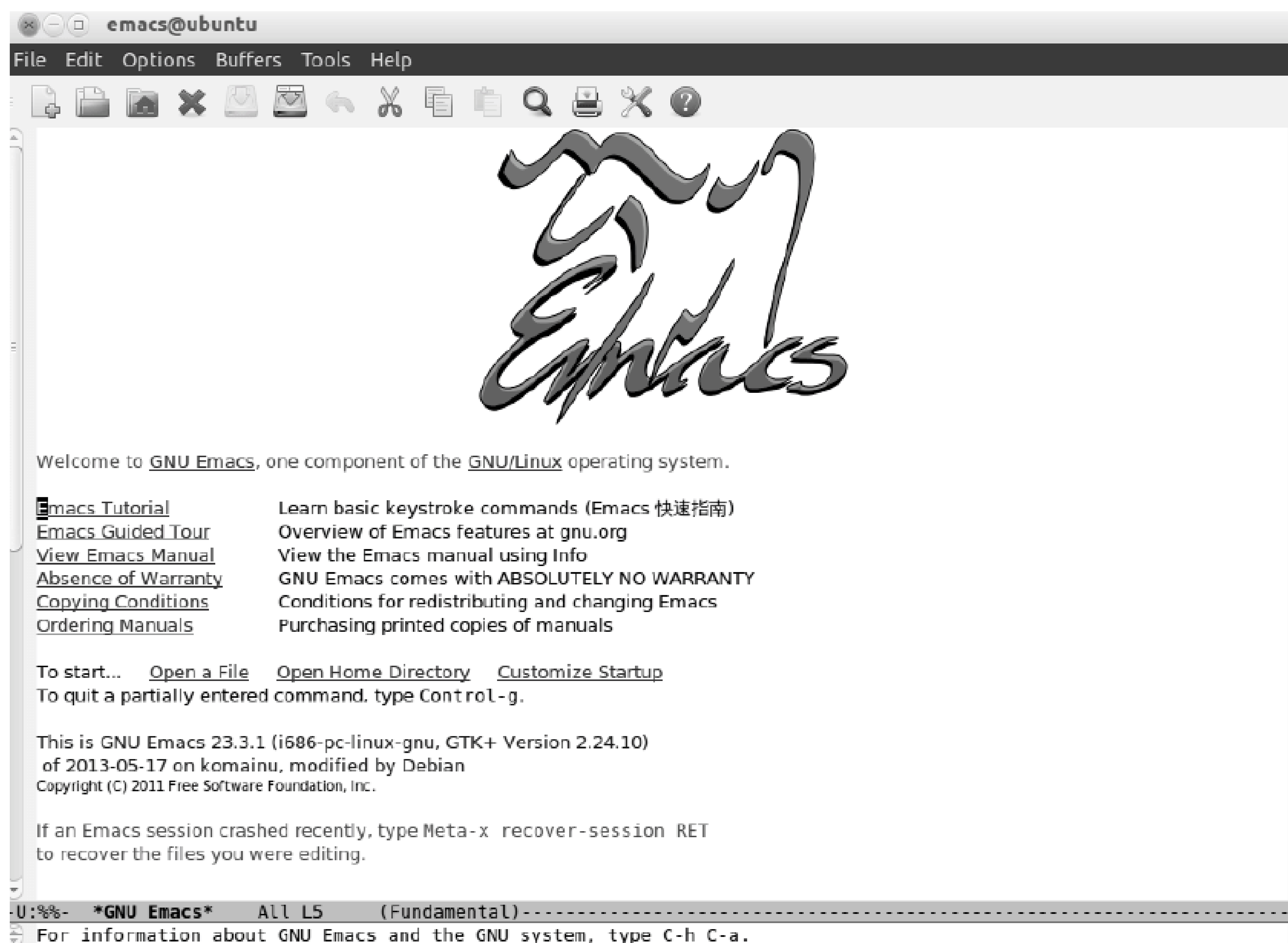


图 2.6 Emacs 的运行界面

Emacs 不仅仅是一个文本编辑器，它更是一个整合环境，或称之为集成开发环境，其具有可移植性强，既可以在图形界面下运行，也可以在命令行界面下运行等特点。

2.3.3 gedit

gedit 是一个在 Linux 的 GNOME 桌面环境下兼容 UTF-8 的纯文本编辑器，使用 GTK+编写而成，因此十分简单易用，并且提供了良好的语法高亮显示功能，对中文支持也很好，支持包括

GB2312、GBK 在内的多种字符编码，其运行界面如图 2.7 所示。

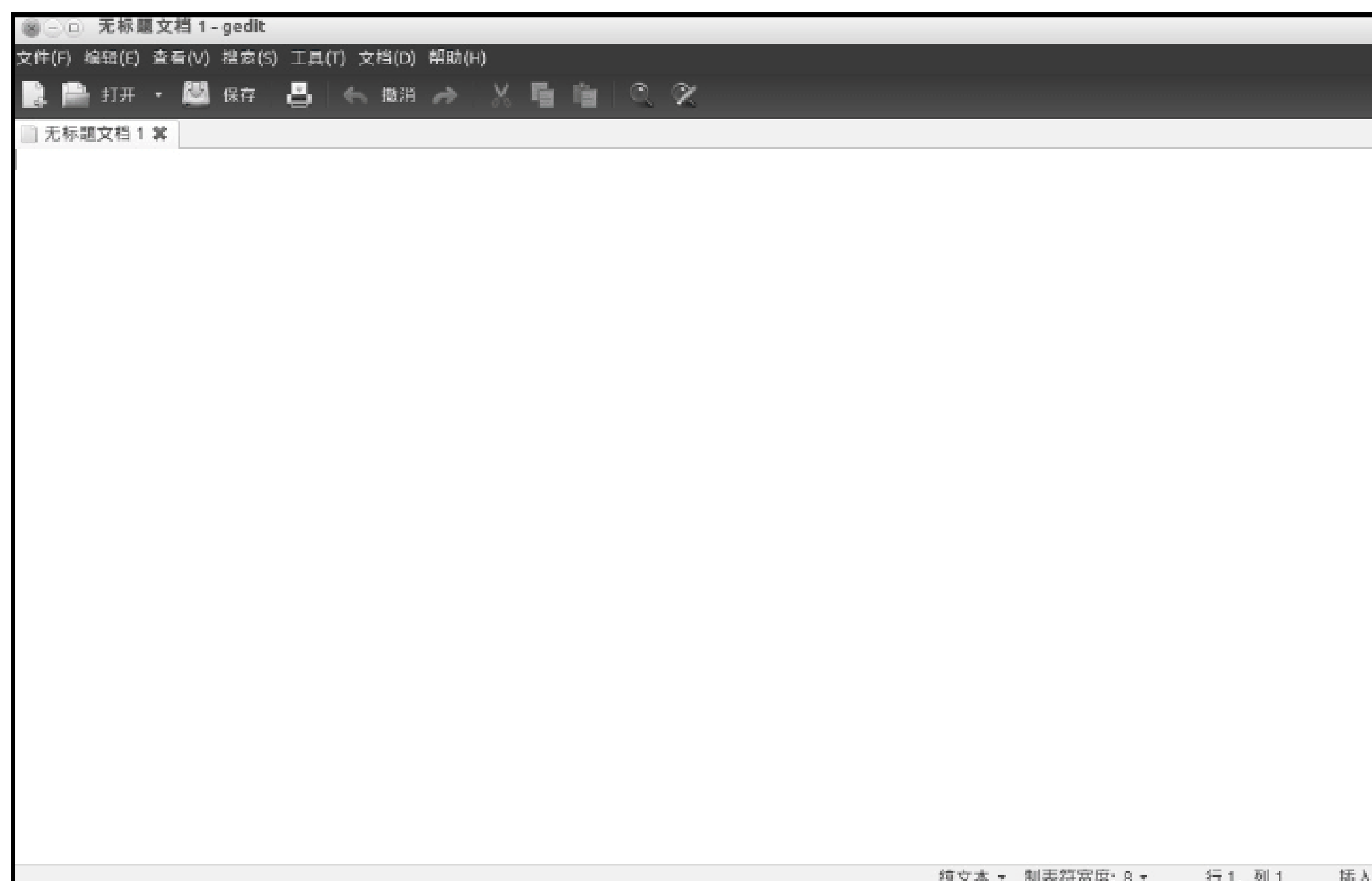


图 2.7　gedit 的运行界面

2.3.4　在 Linux 中编辑 C 语言代码文件的应用实例

本小节通过一个在 Linux 的命令行中进行 C 语言代码编辑的实例来介绍在 Linux 中进行 C 语言代码编辑的过程。

【例 2.1】在 Linux 中编辑 C 语言代码文件

例 2.1 是一个使用 vim 编辑器来编写简单 C 语言代码文件的实例，该 C 语言代码文件是一个简单的调用 printf 函数来输出“This is a gcc test!”字符串并且回车换行的应用，其文件名为 Examhello.c，代码内容如下：

```
#include  <stdio.h>                    //声明库函数
int main(void)
{
    printf("This is a gcc test!\n");   //输出一个字符串
    return 0;
}
```

操作步骤如下：

01 在 Linux 的 shell 终端中使用 vim 命令新建一个名称为 exam201hello.c 的文件，此时 vim 启动并且进入命令行工作模式，如图 2.8 所示。

```
alloy@ubuntu:~/linuxc/chapter2$ vim exam201hello.c
```

图 2.8　启动 vim 并且进入命令行工作模式

02 在命令行工作模式下使用“a”命令快捷键，进入 vim 的编辑模式，输入文件对应的代码，如图 2.9 所示。

图 2.9　vim 的编辑状态

03 使用“Esc”快捷键结束编辑，然后使用“;”快捷键进入底行工作模式，输入“wq”命令保存文件并且退出，此时完成了文件的编辑。

注意

在 gcc 编译过程中，其错误会定义到具体的行号，为了方便查找错误对应的行，可以在底行模式下使用“set nu”命令在每一行前添加行号，添加行号后的文本界面如图 2.10 所示。



图 2.10　添加行号后的文件

2.4　Linux C 语言的编译器 gcc

gcc（GNU C Compiler）是 GNU 推出的功能强大、性能优越的多平台编译器，其可以编译利用 C、C++和 Object C 等语言编写的程序。gcc 编译出的目标代码质量非常好，编译速度也很快，并且 gcc 是一个交叉平台编译器，它能够在当前处理器平台上为多种不同体系结构的硬件平台开发软件，因此尤其适合嵌入式领域的开发编译。

2.4.1　gcc 的安装和配置

在许多 Linux 的发行版（例如 Ubuntu 12.04）中 gcc 是默认安装的，但是其还缺少常用的头文件和库文件，所以还需要安装 build-essential 这个包，可以在联网状态下使用如下命令来安装这个包。

```
alloy@ubuntu:~$ sudo apt-get install build-essential
```

其中，apt-get 是 Ubuntu 下的软件管理命令，它可以安装、删除、更新系统中的软件包。install 是安装软件包，build-essential 是待安装的软件包名称。由于安装软件需要 root 权限，因此，系统会提示输入密码，在输入密码后，系统会自动安装编译所需要的相关文件，系统在安装 build-essential 时，会把程序文件放入以下几个目录。

- /usr/lib：大部分的编译程序放在这个目录。在这里有编译时需要的可执行程序，还有一些特定版本的库文件与头文件等。
- /usr/bin/gcc：指的是编译程序，即实际在命令行中执行的程序。这个目录可供各个版本的 gcc 使用，只要利用不同的编译程序目录来安装就可以了。
- /usr/include：这个目录及其子目录下包含程序所需要的头文件。若缺少头文件，则 gcc 在编译时会出现找不到头文件的错误。

在安装完成之后，可以使用“gcc-v”命令来查看 gcc 的版本号，其执行过程如下：

```
alloy@ubuntu:~$ gcc -v
使用内建 specs。
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/4.6/lto-wrapper
目标：x86_64-linux-gnu
配置为：../src/configure -v --with-pkgversion='Ubuntu/Linaro 4.6.3-1ubuntu5'
--with-bugurl=file:///usr/share/doc/gcc-4.6/README.Bugs --enable-languages=c,c++,fortran,objc,obj-c++
--prefix=/usr --program-suffix=-4.6 --enable-shared --enable-linker-build-id --with-system-zlib
--libexecdir=/usr/lib --without-included-gettext --enable-threads=posix
--with-gxx-include-dir=/usr/include/c++/4.6 --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu
--enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-gnu-unique-object --enable-plugin
--enable-objc-gc --disable-werror --with-arch-32=i686 --with-tune=generic --enable-checking=release
--build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
线程模型：posix
gcc 版本 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)
```



注 意

由于 gcc 仍然处于不断地完善与更新之中，每隔几个月就会有新的稳定发行版本产生，用户可以通过访问 http://www.gnu.org/software/gcc/来了解 gcc 的最近发展，下载最新的软件套件。

2.4.2 gcc 对 C 语言的处理过程

gcc 对 C 语言的处理需要经过如下 4 个步骤：

01 预处理，这是 C 语言处理的第一阶段，此时 gcc 需要对 C 语言源文件中包含的各种头文件和宏定义进行处理，如#define、#include、#if 等。

02 编译，在这个过程中 gcc 根据输入的 C 语言源文件来产生汇编语言，由于通常是立即调用汇编程序，所以其输出一般不保存在文件中。在编译步骤中 gcc 首先检查代码的规范性、是否存在语法错误等，以确定代码的功能，然后将 C 语言代码翻译成汇编语言代码。

03 汇编，gcc 将刚刚得到的汇编语言用于输入，产生具有.o 扩展名的目标文件。

04 链接，在本阶段中各目标文件被 gcc 放在可执行文件的适当位置上，该程序引用的函数也放在可执行文件中（对使用共享库的程序稍有不同）。

下面以在例 2.1 中完成 C 语言源文件的处理过程为例来具体介绍 gcc 的工作过程。

01 预处理阶段：在这个阶段过后会生成预处理文件（后缀名为“.i”），以下为生成的 hello.i 文件的部分内容，可以看到 gcc 把 stdio.h 头文件的部分内容插入到了文件中：

```
typedef int (*__gconv_trans_fct) (struct __gconv_step *,
              struct __gconv_step_data *, void *,
              __const unsigned char *,
              __const unsigned char **,
              __const unsigned char *, unsigned char **,
              size_t *);
…
//以上为预处理阶段插入的 stdio.h 文件部分内容
```

```
# 2 "hello.c" 2
int main()
{
      printf("Hello gcc!\n");
      return 0;
}
```

02 编译阶段：在这个阶段中 gcc 会生成汇编代码文件 hello.s，其部分内容如下。

```
      .file   "hello.c"
        .section    .rodata
        .align 4
.LC0:
        .string             "Hello gcc!"
        .text
.globl main
        .type main, @function
main:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        andl $-16, %esp
        movl $0, %eax
        addl $15, %eax
        addl $15, %eax
        shrl $4, %eax
        sall $4, %eax
        subl %eax, %esp
        subl $12, %esp
        pushl $.LC0
        call puts
        addl $16, %esp
        movl $0, %eax
        leave
        ret
```

03 汇编阶段：在这个阶段 gcc 把编译阶段生成的“.s”文件转换成目标文件。

04 链接阶段：在本阶段中涉及的最关键因素是函数库，从源文件中可以看到在其中并没有定义“printf”的函数实现，且在预编译中包含的“stdio.h”中也只有该函数的声明，而没有定义函数的实现，这是因为系统把这些函数的实现都放到名为 libc.so.6 的库文件中去了，在没有特别指定时，gcc 会到系统默认的搜索路径“/usr/lib”下查找，也就是链接到 libc.so.6 函数库中去，这样就能调用函数“printf”了，而这也正是链接的作用。

函数库有静态库和动态库两种。

- 静态库是指编译链接时，将库文件的代码全部加入可执行文件中，因此生成的文件比较大，但在运行时也就不再需要库文件了，其后缀名通常为“.a”。
- 动态库与之相反，在编译链接时并没有将库文件的代码加入可执行文件中，而是在程序执行时加载库，这样可以节省系统的开销。一般动态库的后缀名为“.so”，如前面所述的 libc.so.6 就是动态库。


注 意

gcc 在编译时默认使用动态库。

2.4.3 gcc 的基础使用方法

和大多数 Linux 下的 shell 命令使用方法类似，gcc 的基本使用格式如下：

```
gcc [选项] 文件名
```

gcc 可以通过选项对程序的生成进行全面控制，每个选项可以有多种取值，在此只对其常用部分进行介绍，其余的参数可以参考 gcc 的手册或其他专门资料，常用选项说明如表 2.7 所示。

表 2.7 gcc 常用选项说明

选项	说明
-c	仅对源文件进行编译，不链接生成可执行文件。在对源文件进行查错或只需产生目标文件时可以使用该选项
-o filename	将经过 gcc 处理过的结果保存为 filename，这个结果文件可以是预处理文件、汇编文件、目标文件或者最终的可执行文件。假设被处理的源文件为 file1，如果这个选项被忽略，那么生成的可执行文件的默认名称为 a.out；目标文件的默认名为 file1.o；汇编文件的默认名为 file1.s；生成的预处理文件发送到标准输出设备 stdout
-g 或-gdb	在可执行文件中加入调试信息，方便进行程序的调试。如果使用“-gdb”选项，表示加入 gdb 扩展的调试信息，以便使用 gdb 来进行调试
-O[0、1、2、3]	对生成的代码进行优化，括号中的部分为优化级别，默认的情况为 2 级优化，0 为不优化。优化和调试通常不兼容，同时使用“-g”和“-O”选项经常会使程序产生奇怪的运行结果，所以不要同时使用“-g”和“-O”选项
-Idir	将 dir 目录加到搜索头文件的目录列表中去，并优先于 gcc 缺省的搜索目录。在有多个“-I”选项的情况下，按命令行上“-I”选项的前后顺序搜索，dir 可使用相对路径
-Ldir	将 dir 目录加到搜寻“-L”选项指定的函数库文件的目录列表中去，并优先于 gcc 缺省的搜索目录。在有多个“-L”选项的情况下，按命令行上“-L”选项的前后顺序搜索，dir 可使用相对路径
-lname	在链接时使用函数库 name.a，链接程序在“-Ldir”选项指定的目录下，以及“/lib”，“/usr/lib”目录下寻找该库文件。在没有使用“-static”选项时，如果发现共享函数库 name.so，则使用 name.so 进行动态链接

gcc 的命令选项可以组合使用，不过在使用时，每个命令选项都要有一个自己的连字符“-”。如果采用简写的方式，很可能使命令的含义完全不同。

在 Linux 下生成的可执行文件没有固定的扩展名。任何符合 Linux 要求的文件名，只要文件的访问属性中有可以执行的属性，该文件就是可以执行的，因此，在使用上面介绍的“-o filename”参数时，如果是生成链接后的可执行文件，filename 变量可以取任意一个符合 Linux 要求的文件名。

gcc 命令中的第 2 部分是一个输入给 gcc 命令的文件，gcc 按照命令选项的要求对输入文件进行处理，形成结果输出文件。输入的文件不一定是 C 的源代码文件，还可能是预处理文件、目标文件等，gcc 通过输入文件的扩展名来确定输入文件类型，表 2.8 列出了 gcc 支持的与 C 语言相关的输入文件类型。

表 2.8　gcc 支持的与 C 语言相关的输入文件类型

扩展名	类型
.c	C 语言源程序，可以被 gcc 预处理、编译、汇编、链接
.C、.cc、.cp、.cpp、.c++、.cxx	C++语言源程序，可以被 gcc 预处理、编译、汇编、链接
.i	预处理后的 C 语言源程序，可以被 gcc 编译、汇编、链接
.ii	预处理后的 C++语言源程序，可以被 gcc 编译、汇编、链接
.s	预处理后的汇编程序，可以被 as 汇编、链接
.S	未预处理的汇编程序，可以被 as 预处理、汇编、链接
.h	头文件，不进行任何操作
.o	编译后的目标文件，传送给 ld
.a	目标文件库，传送给 ld

2.4.4　gcc 的应用实例

本小节通过两个不同的 gcc 编译实例来介绍 gcc 的具体使用方法。

【例 2.2】gcc 编译器应用实例（一）

本实例是使用 gcc 对在例 2.1 中建立的 C 语言源文件进行编译的过程，对其详细步骤说明如下：

01 在 Linux 的 shell 中使用如下 gcc 命令对这个文件进行编译：

```
alloy@ubuntu:~/linuxc/chapter2$ gcc exam201hello.c -o exam201hello
```

02 此时可以看到在对应的目录下生成了 exam201hello 可执行文件（通常会以绿色在终端中显示），运行后可以看到对应的输出，整个执行过程如下：

```
alloy@ubuntu:~/linuxc/chapter2$ gcc exam201hello.c -o exam201hello
//使用 gcc 对.c 文件进行编译
alloy@ubuntu:~/linuxc/chapter2$ ./exam201hello          //执行刚刚生成的可执行文件
This is a gcc test!                                      //可执行文件的输出
```

在实际的开发过程中经常遇到应用代码比较复杂的情况，此时通常采用将主函数和其他函数放在不同文件中的方法。除了主程序之外，每个函数都由函数声明（函数头）和函数实现（函数体）两部分组成。函数的声明一般放在头文件（.h）中，而函数的定义文件放在实现文件中（.c），gcc 可以很容易地把多个源文件编译成目标代码并进行链接，如例 2.3 所示。

【例 2.3】gcc 编译器应用实例（二）

这是使用另外一个 C 语言文件来存放一个输出函数，然后使用 gcc 对多个 C 语言文件进行一

次性编译的过程，操作步骤如下。

01 在当前工作目录下建立一个名称为 Examhellosun.c 的 C 语言文件，其内容如下：

```
#include <stdio.h>
void sunprintf(void)
{
    printf("this is a test from anthor .c!\n");
}
```

02 然后再建立一个名称为 Examhello.h 的.h 头文件，其内容如下：

```
void sunprintf(void);
```

03 随后建立一个名称为 Examhello.c 的 C 语言文件，在其中声明并且引用头文件 Examhello.h，其内容如下：

```
#include <stdio.h>
#include "Examhello.h"
int main(void)
{
  printf("This is a gcc test!\n");
  sunprintf();
  return 0;
}
```

04 此时可以使用如下的命令来对这两个 C 语言文件进行编译，并且链接生成可执行文件 Examhello。

```
alloy@ubuntu:~/chapter2$ gcc Examhello.c Examhellosun.c -o Examhello
```

05 执行 Examhello 文件后可以看到如下的输出，其中第 2 行是 Examhellosun.c 文件中的 sunprintf 函数输出。

```
This is a gcc test!
this is a test from anthor .c!
```

2.5 Linux C 语言的调试工具 gdb

在实际开发过程中，C 语言的代码除了符合最基本的语法规范之外，还必须符合设计者的逻辑意图，如果发现生成的可执行文件运行结果不正确，则可以通过相应的调试环境来跟踪调试，Linux 提供了一个称为 gdb 的调试程序，其是 GNU 开发并发布的 UNIX/Linux 下的程序调试工具，能在程序运行时观察程序的内部结构和内存的使用情况，主要提供以下一些功能：

- 监视程序中变量的值。
- 设置断点以使程序在指定的代码行上停止执行。
- 一行行地执行代码。

2.5.1　gdb 的基础使用

通常来说 gdb 是 Linux 在安装时自带的，在命令行上键入“gdb”字符串并按回车键则会启动 gdb 调试环境，在屏幕上会看到如下类似内容（根据 Linux 的发行版本以及 gdb 的版本差别会有少许区别）：

```
alloy@ubuntu:~$ gdb
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>.
```

gdb 是一个功能强大的调试器，其支持的调试命令非常丰富，可以实现不同的功能。这些命令包括从简单的文件装入到允许检查所调用的堆栈内容的复杂命令。表 2.9 列出了使用 gdb 调试时会用到的一些常用命令，如果想进一步了解 gdb 的详细使用，可以参考 gdb 的帮助文档。

表 2.9　gdb 的基本命令

命令	说明
file	装入想要调试的可执行文件
kill	终止正在调试的程序
list	列出产生执行文件的部分源代码
next	执行一行源代码但不进入函数内部
step	执行一行源代码而且进入函数内部
run	执行当前被调试的程序
quit	退出 gdb
watch	动态监视一个变量的值
make	不退出 gdb 而重新产生可执行文件
call name(args)	调用并执行名为 name、参数为 args 的函数
return value	停止执行当前函数，并将 value 返回给调用者
break	在代码里设置断点，使程序执行到此处被挂起

通常来说，调用 gdb 只需要使用一个参数，其标准格式如下：

```
gdb <可执行程序名>
```

如果程序运行时产生了错误，会在当前目录下产生核心内存映像 core 文件，可以在指定执行文件的同时为可执行程序指定一个 core 文件：

```
gdb <可执行文件名> core
```

除此之外，还可以为要执行的文件指定一个进程号：

```
gdb <可执行文件名> <进程号>
```

以下是使用 gdb 来为例 2.2 所生成的可执行文件指定进程号的过程，gdb 首先会寻找一个文件名为 2000 的文件，如果找不到，则把调试程序的进程号（PID）设成 2000，整个执行过程如下：

```
alloy@ubuntu:~/linuxc/chapter2$ gdb exam201hello 2000
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
……   //此处省略部分内容
Reading symbols from /home/alloy/linuxc/chapter2/exam201hello...(no debugging symbols found)...done.
Attaching to program: /home/alloy/linuxc/chapter2/exam201hello, process 2000
ptrace: 没有那个进程.
/home/alloy/linuxc/chapter2/2000: 没有那个文件或目录.
(gdb)
```

当 gdb 运行时，把任何一个不带选项前缀的参数都作为一个可执行文件、core 文件或被调试程序相关联的进程号。不带任何选项前缀的参数和前面加了“-se”或“-c”选项的参数效果相同。gdb 把第一个前面没有选项说明的参数看作前面加了“-se”选项，也就是需要调试的可执行文件并从此文件里读取符号表，如果有第 2 个前面没有选项说明的参数，将被看作是跟在“-c”选项后面，也就是需要调试的 core 文件名。

如果不希望看到 gdb 开始的提示信息，可以用“gdb –silent”执行调试工作，通过更多的选项，开发者可以按自己的喜好定制 gdb 的行为。

输入“gdb –help”或“-h”可以得到 gdb 启动时的所有选项提示。gdb 命令行中的所有参数都按照排列的顺序传给 gdb，除非使用了“-x”参数。

gdb 的许多选项都可以用缩写形式代表，这可以利用“-h”查看。在 gdb 中也可以采取任意长度的字符串代表选项，只要保证 gdb 能唯一地识别此参数就行。

表 2.10 列出了 gdb 中一些最常用的参数选项。

表 2.10　gdb 中常用的参数选项

选项	说明
-s filename	从 filename 指定的文件中读取要调试的程序符号表
-e filename	执行 filename 指定的文件，并通过与 core 文件进行比较来检查正确的数据
-se filename	从 filename 中读取符号表并作为可执行文件进行调试
-c filename	把 filename 指定的文件作为一个 core 文件
-c num	把数字 num 作为进程号和调试的程序进行关联，与 attach 命令相似
-command filename	按照 filename 指定的文件中的命令执行 gdb 命令，在 filename 指定的文件中存放着一系列的 gdb 命令，就像一个批处理
-d path	指定源文件的路径，把 path 加入到搜索源文件的路径中
-r	从符号文件中一次读取整个符号表，而不是使用默认的方式首先调入一部分符号，当需要时再读入其他部分，这会使 gdb 的启动较慢，但可以加快以后的调试速度

2.5.2 gdb 运行模式的选择

gdb 提供了包括“批模式”或“安静模式”在内的一系列运行模式，可以通过 gdb 运行时在命令行中通过选项来选择，表 2.11 列出了 gdb 运行模式的相关选项。

表 2.11 gdb 运行模式选项

选项	说明
-n	不执行任何初始化文件中的命令（一级初始化文件称为.gdbinit）。一般情况下在这些文件中的命令会在所有的命令行参数都被传给 gdb 后执行
-q	设定 gdb 的运行模式为“安静模式”，可以不输出介绍和版权信息，这些信息在“批模式”中也不会显示
-batch	设定 gdb 的运行模式为“批模式”，gdb 在“批模式”下运行时，会执行命令文件中的所有命令，当所有命令都被成功执行后 gdb 返回状态 0，如果在执行过程中出错，gdb 返回一个非零值
-cd dir	把 dir 作为 gdb 的工作目录，而非当前目录（gdb 缺省时把当前目录作为工作目录）

2.5.3 gdb 应用实例

例 2.4 是一个使用 gdb 对例 2.2 生成的可执行文件进行调试的应用实例。

【例 2.4】gdb 编译器应用实例

01 运行“gdb + 待调试的可执行文件名称”命令来启动调试。

02 使用“b”快捷键在程序开始处设置断点，然后使用“run”开始调试。

03 使用“n”快捷键即可执行下一条语句，其间还可以使用其他命令来观察相应的变量运行情况。

以上操作的执行过程如下：

```
alloy@ubuntu:~/linuxc/chapter2$ gdb exam201hello          //启动 gdb 对 exam201hello 进行调试
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
……   //此处省略部分内容
Reading symbols from /home/alloy/linuxc/chapter2/exam201hello...(no debugging symbols found)...done.
(gdb) b main                              //在 main 函数处设置一个断点
Breakpoint 1 at 0x4004f8                  //设置断点完成
(gdb) run
Starting program: /home/alloy/linuxc/chapter2/exam201hello

Breakpoint 1, 0x00000000004004f8 in main ()       //断点停止
(gdb) n                                            //执行下一条语句
Single stepping until exit from function main,
which has no line number information.
This is a gcc test!
0x00007ffff7a3b78d in __libc_start_main () from /lib/x86_64-linux-gnu/libc.so.6
(gdb) n                                             //执行下一条语句
```

```
Single stepping until exit from function __libc_start_main,
which has no line number information.
[Inferior 1 (process 4731) exited normally]
```


注 意

关于使用 gdb 对 C 语言生成的可执行文件的更多调试技巧，读者可以参阅 gdb 的相关资料，在此不再赘述。

2.6 Linux C 语言的项目管理工具 make

在实际应用中一个 C 语言的工程项目常常由多个文件组成，此时为了对多个文件进行管理和处理，可以使用 make 项目管理器。

2.6.1 make 项目管理器的基础

使用项目管理器的主要目的是用于管理较多的文件，在第 2.1 小节中已介绍过 C 语言文件的编译过程分为编译、汇编、链接阶段，其中编译阶段仅检查语法错误以及函数与变量是否被正确地声明了，在链接阶段则主要完成函数链接和全局变量的链接，因此，那些没有改动的源代码根本不需要重新编译，只要把它们重新链接进去就可以了，所以用户需要有一个项目管理器能够自动识别更新的文件代码，而不需要重复输入冗长的命令行，这时，make 工程管理器就应运而生了。

实际上，make 工程管理器也就是个“自动编译管理器”，这里的“自动”是指它能够根据文件时间戳自动发现更新过的文件而减少编译的工作量，同时其通过读入 makefile 文件的内容来执行大量的编译工作。用户只需编写一次简单的编译语句就可以了，所以大大提高了实际项目的工作效率。

1. make 的基本结构

makefile 是 make 项目管理器中使用的配置文件，其通常由以下几个部分组成。

- 目标体：make 项目管理器生成的目标文件（target）或者可执行文件。
- 依赖文件：make 项目管理器创建目标体所需要的文件（dependency file），通常是 C 语言文件、C 语言的头文件等。
- 相关操作命令：make 项目管理器使用依赖文件来创建目标体所需要的命令（command），这些操作命令必须以制表符（Tab）开头。

一个标准的 makefile 文件的写法如例 2.5 所示。

【例 2.5】一个标准的 makefile 文件

两个 makefile 文件分别命名为 hello.c 和 hello.h 的文件经过编译生成目标体 hello.o，执行的命令为 gcc 编译指令：gcc –c hello.c。

实例的应用代码如下：

```
#The simplest example
hello.o: hello.c hello.h
```

```
            gcc –c hello.c –o hello.o
```

此时可以使用 make 项目管理器了，其标准调用格式如下：

```
make target
```

运行时 make 项目管理器会自动读入 makefile 文件，找到相关的依赖文件并执行对应 target 的 command 语句，可以看到以上 makefile 的文件输出结果如下：

```
alloy@ubuntu:/$ make hello.o
gcc –c hello.c –o hello.o
alloy@ubuntu:/$ ls
hello.c    hello.h    hello.o    makefile
```

可以看到，make 项目管理器执行了“hello.o”对应的命令语句，并生成了“hello.o”目标体。

2. make 的变量

通常来说使用如例 2.5 所示的简单 makefile 文件是没有意义的，在实际应用中 makefile 通常都包括了大量的依赖文件和操作命令，例 2.6 是一个较为复杂的 makefile 文件示例。

【例 2.6】一个较为复杂的 makefile

在这个 makefile 中有 3 个目标体（target），分别为 exam1、exam2.o 和 exam3.o，其中第一个目标体的依赖文件就是后两个目标体，如果用户使用命令“make exam1”，则 make 管理器找到 exam1 目标体开始执行。

实例的应用代码如下：

```
exam1:exam2.o exam3.o
        gcc exam2.o bar.o -o myprog
exam2.o : exam2.c exam2.h head.h
       gcc –Wall –O -g –c exam2.c -o exam2.o
exam3.o : bar.c head.h
       gcc - Wall –O -g –c yul.c -o exam3.o
```

在执行以上 makefile 文件时，make 项目管理器会自动检查相关文件的时间戳，首先在检查“exam2.o”、“exam3.o”和“exam1”3 个文件的时间戳之前，它会向下查找那些把“exam2.o”或“exam3.o”作为目标文件的时间戳。例如，“exam2.o”的依赖文件为“exam2.c”、“exam2.h”、“head.h”。如果这些文件中任何一个时间戳比“exam2.o”新，则将会执行命令“gcc －Wall －O -g －c exam2.c -o exam2.o”，从而更新文件“exam2.o”。在更新完“exam2.o”或“exam3.o”之后，make 会检查最初的“exam2.o”、“exam3.o”和“exam1”3 个文件，只要文件“exam2.o”或“exam3.o”中至少有一个文件的时间戳比“exam1”新，则第二行命令就会被执行。这样，make 就完成了自动检查时间戳的工作，开始执行编译工作，这也就是 make 工作的基本流程。

接下来，为了进一步简化编辑和维护 makefile 文件，make 允许在 makefile 文件中创建和使用变量，变量是在 makefile 文件中定义的名字，用来代替一个文本字符串，该文本字符串称为该变量的值。在具体要求下，这些值可以代替目标体、依赖文件、命令以及 makefile 文件中的其他部分。

在 makefile 文件中的变量定义有两种方式：一种是递归展开方式，另一种是简单方式。

递归展开方式定义的变量是在引用该变量时进行替换的，即如果该变量包含了对其他变量的引用，则在引用该变量时一次性地将内嵌的变量全部展开，虽然这种类型的变量能够很好地完成用户的指令，但是它也有严重的缺点，例如不能在变量后追加内容（因为语句“CFLAGS = $(CFLAGS) –O”在变量扩展过程中可能导致无限循环）。

为了避免上述问题，简单扩展型变量的值在定义处展开，并且只展开一次，因此它不包含任何对其他变量的引用，从而消除变量的嵌套引用。

- 递归展开方式的定义格式为：VAR=var。
- 简单扩展方式的定义格式为：VAR:=var。
- make 中的变量均使用的格式为：$(VAR)。


注 意

makefile 的变量名是不包括“:”、“#”、“=”以及结尾空格的任何字符串。同时，应尽量避免变量名中包含字母、数字以及下划线以外的情况，因为它们可能在将来被赋予特别的含义。变量名是大小写敏感的，例如变量名“foo”、“FOO”、和“Foo”代表不同的变量。推荐在 makefile 内部使用小写字母作为变量名，预留大写字母作为控制隐含规则参数或用户重载命令选项参数的变量名。

【例 2.7】使用变量的 makefile 文件

例 2.7 是使用变量重写例 2.6 的 makefil 文件 e，其中使用 OBJS 代替 exam2.o 和 exam3.o，用 CC 代替 gcc，用 CFLAGS 代替“-Wall -O –g”。这样在以后修改时，就可以只修改变量定义，而不需要修改下面的定义实体，从而大大简化了 makefile 文件的维护工作量。

实例的应用代码如下：

```
OBJS = exam2.o exam3.o
CC = gcc
CFLAGS = -Wall -O -g
exam1 : $(OBJS)
        $(CC) $(OBJS) -o exam1
exam2.o : exam2.c exam2.h
        $(CC) $(CFLAGS) -c exam2.c -o exam2.o
exam3.o : yul.c yul.h
        $(CC) $(CFLAGS) -c yul.c -o exam3.o
```

makefile 中的变量分为用户自定义变量、预定义变量、自动变量及环境变量。如上例中的 OBJS 就是用户自定义变量，自定义变量的值由用户自行设定，而预定义变量和自动变量为通常在 makefile 中都会出现的变量，它们的一部分有默认值，也就是常见的设定值，当然用户可以对其进行修改。

预定义变量包含了常见编译器、汇编器的名称及其编译选项，表 2.12 列出了 makefile 中常见预定义变量及其部分默认值。

表 2.12　makefile 的常见变量及其默认值

预定义变量	含义
AR	库文件维护程序的名称，默认值为 ar
AS	汇编程序的名称，默认值为 as
CC	C 编译器的名称，默认值为 cc
CPP	C 预编译器的名称，默认值为“$(CC) –E”
CXX	C++编译器的名称，默认值为 g++
FC	Fortran 编译器的名称，默认值为 f77
RM	文件删除程序的名称，默认值为“rm –f”
ARFLAGS	库文件维护程序的选项，无默认值
ASFLAGS	汇编程序的选项，无默认值
CFLAGS	C 编译器的选项，无默认值
CPPFLAGS	C 预编译的选项，无默认值
CXXFLAGS	C++编译器的选项，无默认值
FFLAGS	Fortran 编译器的选项，无默认值

可以看出，上例中的 CC 和 CFLAGS 是预定义变量，其中由于 CC 没有采用默认值，因此，需要把“CC=gcc”明确列出来。

由于常见的 gcc 编译语句中通常包含了目标文件和依赖文件，而这些文件在 makefile 文件中的目标体所在行中已经有所体现，因此，为了进一步简化 makefile 的编写，就引入了自动变量。自动变量通常可以代表编译语句中出现目标文件和依赖文件等，并且具有本地含义（即下一语句中出现的相同变量代表的是下一语句的目标文件和依赖文件）。表 2.13 列出了 makefile 中常见的自动变量。

表 2.13　makefile 中常见的自动变量

自动变量	含义
$*	不包含扩展名的目标文件名称
$+	所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
$<	第一个依赖文件的名称
$?	所有时间戳比目标文件晚的依赖文件，并以空格分开
$@	目标文件的完整名称
$^	所有不重复的依赖文件，以空格分开
$%	如果目标是归档成员，则该变量表示目标的归档成员名称

【例 2.8】使用自动变量的 makefile 文件

利用自动变量可以把例 12.7 改写为例 12.8。

实例的应用代码如下：

```
OBJS = exam2.o exam3.o
```

```
CC = gcc
CFLAGS = -Wall -O -g
exam1 : $(OBJS)
        $(CC) $^ -o $@
exam2.o : exam2.c exam2.h
        $(CC) $(CFLAGS) -c $< -o $@
exam3.o : yul.c yul.h
        $(CC) $(CFLAGS) -c $< -o $@
```


注 意

在 makefile 文件中还可以使用环境变量。使用环境变量的方法相对比较简单，make 在启动时会自动读取系统当前已经定义了的环境变量，并且会创建与之具有相同名称和数值的变量，但是，如果用户在 makefile 中定义了相同名称的变量，那么用户自定义变量将会覆盖同名的环境变量。

3. make 的规则

makefile 的规则是 make 进行处理的依据，它包括了目标体、依赖文件及其之间关系的命令语句。在上面的例子中，都显式地指出了 makefile 中的规则关系，如“$(CC) $(CFLAGS) -c $< -o $@”，但为了简化 makefile 的编写，make 工程管理器还定义了隐式规则和模式规则。

（1）隐式规则

隐式规则能够告诉 make 怎样使用传统的规则完成任务，当用户使用其时就不必详细指定编译的具体细节，而只需把目标文件列出即可，此时 make 工程管理器会自动搜索隐式规则目录来确定如何生成目标文件，如例 2.8 可以改写为如下形式：

```
OBJS = exam2.o exam3.o
CC = gcc
CFLAGS = -Wall -O -g
exam1 : $(OBJS)
        $(CC) $^ -o $@
```

由于在 makefile 的隐式规则中指出所有“.o”文件都可自动由“.c”文件使用命令“$(CC) $(CPPFLAGS) $(CFLAGS) -c file.c –o file.o”来生成，因此“exam2.o”和“exam3.o”就会分别通过调用“$(CC) $(CFLAGS) -c exam2.c -o exam2.o”和“$(CC) $(CFLAGS) -c yul.c -o exam3.o”来生成，表 2.14 是常见的隐式规则目录。

表 2.14　常见的隐式规则目录

对应语言后缀名	隐式规则
C 编译：“.c”变为“.o”	$(CC) –c $(CPPFLAGS) $(CFLAGS)
C++编译：“cc”或“.C”变为“.o”	$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)
Pascal 编译：“.p”变为“.o”	$(PC) -c $(PFLAGS)
Fortran 编译：“.r”变为“-o”	$(FC) -c $(FFLAGS)

（2）模式规则

模式规则用来定义相同处理规则的多个文件。它不同于隐式规则，隐式规则仅能够利用 make 默认的变量来进行操作，而模式规则还能引入用户的自定义变量，为多个文件建立相同的规则，从而简化 makefile 的编写。

模式规则的格式类似于普通规则，这个规则中的相关文件前必须用“%”标明。使用模式规则修改后的 makefile 文件的示例如下：

```
OBJS = exam2.o exam3.o
CC = gcc
CFLAGS = -Wall -O -g
exam1 : $(OBJS)
	$(CC) $^ -o $@
%.o : %.c
	$(CC) $(CFLAGS) -c $< -o $@
```

2.6.2　make 项目管理器的使用

make 项目管理器的使用非常简单，只需在 make 命令的后面键入目标名即可建立指定的目标，如果直接运行 make，则建立 makefile 中的第一个目标。此外 make 还具有丰富的命令行选项，可以完成各种不同的功能，这些命令行如表 2.15 所示。

表 2.15　make 项目管理器的命令行

命令格式	含义
-C dir	读入指定目录下的 makefile
-f file	读入当前目录下的 file 文件作为 makefile
-i	忽略所有的命令执行错误
-I dir	指定被包含的 makefile 所在目录
-n	只打印要执行的命令，但不执行这些命令
-p	显示 make 变量数据库和隐式规则
-s	在执行命令时不显示命令
-w	如果 make 在执行过程中改变目录，则打印当前目录名

对于简单的项目而言用户可以自行编写 makefile 文件，对于较大的项目则可以使用 autotools 来自动生成 makefile，需要注意的是在使用之前需要安装该工具。

```
alloy@ubuntu:/$ sudo apt-get install autoconf
```

autotools 是一个系列工具，使用 autotools 的过程就是使用这些系列工具的脚本文件来生成最后的 makefile 的过程：

- aclocal：生成一个名称为 aclocal.m4 的用于处理本地宏定义的文件。
- autoscan：在给定目录及其子目录树中检查源文件，若没有给出目录，就在当前目录及其子目录树中进行检查。它会搜索源文件以寻找一般的移植性问题并创建一个文件

"configure.scan"，该文件就是接下来 autoconf 要用到的"configure.in"原型。

- autoconf：对 configure.in 脚本配置文件进行处理。
- autoheader：其负责生成 config.h.in 文件。该工具通常会从"acconfig.h"文件中复制用户附加的符号定义。
- automake：其要用到的脚本配置文件是 makefile.am，用户需要自己创建相应的文件，然后利用 automake 工具转换成 makefile.in，此时运行 configure 自动配置设置文件即可将该.in 文件生成 makefile 文件。

使用 autotools 工具生成 makefile 文件的流程如图 2.11 所示。

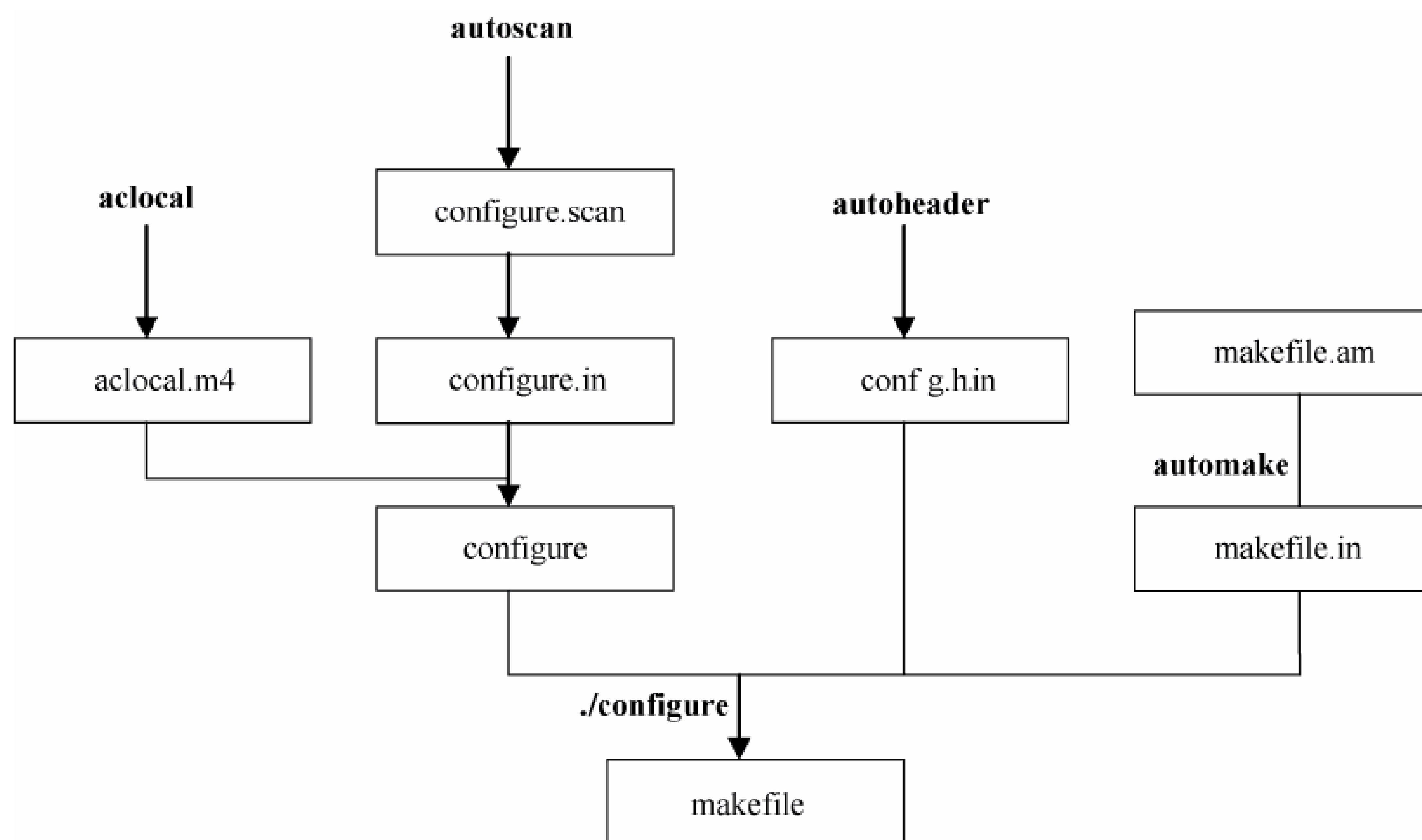


图 2.11　使用 autotools 工具生成 makefile 文件

使用 autotools 生成的 makefile 文件除了能完成编译操作之外，还可以完成其他的操作：

- 使用 make 命令默认执行"make all"操作，可以生成对应的可执行文件。
- 使用 make install 命令会将生成的可执行文件安装到系统目录中（通常为 usr/local/bin 目录）并添加对应的全局变量，此时在任意路径下可以对该可执行文件进行操作。
- 使用 make clean 命令会清除之前所编译的可执行文件和目标文件。
- 使用 make dist 命令会将可执行文件和涉及的文件生成一个压缩文件包（tar.gz）。

2.6.3　make 项目管理器的应用实例

例 2.9 给出了一个使用 autotools 工具来为 hello.c 的 C 语言文件生成 makefile 文件并且使用的实例，其详细步骤描述如下。

【例 2.9】使用 autotools 生成 make 项目文件

首先运行 autoscan（可能需要 sudo 权限），其会搜寻指定目录（默认是当前目录及其子目录）中的源文件，并且创建 configure.scan 文件。

```
alloy@ubuntu:/$sudo autoscan
```

```
autom4te: configure.ac: no such file or directory
autoscan: /usr/bin/autom4te failed with exit status: 1
alloy@ubuntu:/$ls
autoscan.log    configure.scan    hello.c
```

autoscan 会尝试读入“configure.ac”（同 configure.in 的配置文件）文件，此时还没有创建该配置文件，于是它会自动生成一个“configure.in”的原型文件“configure.scan”，该文件和源文件 hello.c 位于同一个子目录下，该.scan 文件的内容可以使用 cat 命令查看，输出如下：

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.59)
#The next one is modified by exam1
#AC_INIT(FULL-PACKAGE-NAME,VERSION,BUG-REPORT-ADDRESS)
AC_INIT(hello,1.0)
# The next one is added by exam1
AM_INIT_AUTOMAKE(hello,1.0)
AC_CONFIG_SRCDIR([hello.c])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_CONFIG_FILES([makefile])
AC_OUTPUT
```

其中对各个部分的说明如下：

- 以“#”号开始的行是注释。
- AC_PREREQ 宏用于声明本文件要求的 autoconf 版本，如本例使用的版本为 2.59。
- AC_INIT 宏用来定义软件的名称和版本等信息，在本例中省略了 BUG-REPORT-ADDRESS，一般为作者的 E-mail。
- AM_INIT_AUTOMAKE 是笔者另加的，它是 automake 所必备的宏，使 automake 自动生成 makefile，PACKAGE 是所要产生软件套件的名称，VERSION 是版本编号。
- AC_CONFIG_SRCDIR 宏用来检查所指定的源代码文件是否存在，以及确定源代码目录的有效性。此处的源代码文件为当前目录下的 hello.c。
- AC_CONFIG_HEADER 宏用于生成 config.h 文件，以便 autoheader 使用。
- AC_CONFIG_FILES 宏用于生成相应的 makefile 文件。
- 中间的注释之间可以分别添加用户测试程序、测试函数库、测试头文件等宏定义。

运行 aclocal 生成一个名称为 aclocal.m4 的文件，用于处理相应的宏定义；然后再运行 autoconf 以生成 configure 可执行文件。

```
alloy@ubuntu:/$aclocal
```

```
alloy@ubuntu:/$autoconf
alloy@ubuntu:/$ls
aclocal.m4   autom4te.cache   autoscan.log   configure   configure.in   hello.c
```

运行 autoheader 生成 config.h.in 文件。

```
alloy@ubuntu:/$autoheader
```

创建一个如下的脚本配置文件 amkefile.am，其中对各个选项的说明如下：

```
AUTOMAKE_OPTIONS=foreign
bin_PROGRAMS= hello
hello_SOURCES= hello.c
```

- 其中的 AUTOMAKE_OPTIONS 为设置 automake 的选项。GNU 对自己发布的软件有严格的规范，例如必须附带许可证声明文件 COPYING 等，否则 automake 执行时会报错。automake 提供了 3 种软件等级：foreign、gnu 和 gnits，可让用户选择采用，默认等级为 gnu。在本示例中采用 foreign 等级，它只检测必须的文件。
- bin_PROGRAMS 定义要产生的执行文件名。如果要产生多个执行文件，每个文件名将利用空格隔开。
- hello_SOURCES 用于定义“hello”这个执行程序所需要的原始文件。如果“hello”这个程序是由多个原始文件产生的，则必须把它用到的所有原始文件都列出来，并用空格隔开。例如：若目标体“hello”需要“hello.c”、“exam1.c”、“hello.h” 3 个依赖文件，则定义“hello_SOURCES=hello.c exam1.c hello.h”。需要注意的是，如果要定义多个执行文件，则对每个执行程序都要定义相应的 file_SOURCES。

使用“automake-a”命令来自动添加一些脚本并且生成 configure.in 文件，容纳后使用 ls 命令来查看生成的文件。

```
alloy@ubuntu:/$automake –a
configure.in: installing './install-sh'
configure.in: installing './missing'
makefile.am: installing 'depcomp'
alloy@ubuntu:/$ ls
aclocal.m4        autoscan.log   configure.in   hello.c       makefile.am   missing
autom4te.cache   configure        depcomp        install-sh   makefile.in   config.h.in
```

运行 configure 将 makefile.in 文件生成最终的 makefile 文件，可以看到，在运行 configure 时收集了系统的信息，用户可以在 configure 命令中对其进行方便地配置。

```
alloy@ubuntu:/$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output file name... a.out
```

```
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
configure: createing ./config.status
config.status: createing makefile
config.status: executing depfiles commands
```

2.7　Linux 中的 C 语言应用代码

在 Linux 操作系统中进行 C 语言代码开发，必须对 Linux 系统有足够的了解，包括代码的运行机制、内存的分配机制、系统调用和库函数等。

2.7.1　C 语言代码的运行机制

Linux 中的程序是一个在磁盘上的可执行文件，内核调用一个 exec 函数将这个可执行文件调入存储器中然后执行之，这个程序的执行实例被称为进程，在 Linux 中每个进程都对应一个唯一的非负数字标识符，称为进程 ID。

对于一个进程而言，其有 8 种方式可以使得其终止，对这些方式的说明如下：

- 在 main 函数中使用 return 语句返回。
- 调用 exit 函数终止进程。
- 调用_exit_或者_Exit 函数终止进程。
- 最后一个线程从其启动例程返回。
- 最后一个线程调用了 pthread_exit 函数。
- 调用 abort 函数。
- 接到一个信号并且终止。
- 最后一个线程对取消请求做出了响应。

这些方式的前 5 种为正常终止一个进程，后三种是异常终止，图 2.12 是 Linux 操作系统启动和终止一个应用程序的示意图。

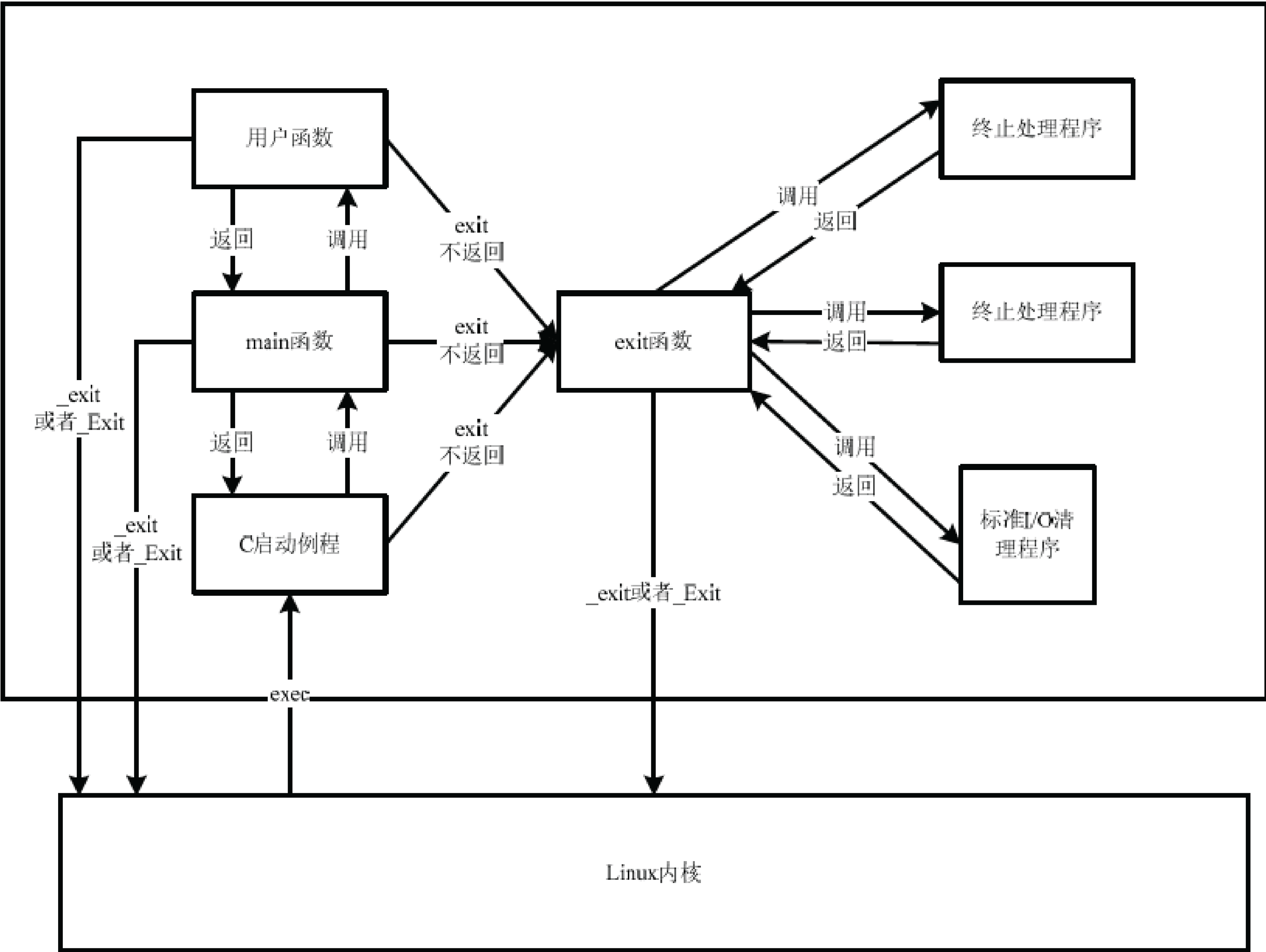


图 2.12　Linux 下的代码运行

总之，在 Linux 操作系统中，内核使程序执行的唯一方法是调用一个 exec 函数，进程自愿终止的唯一方法是显式或者隐式地调用_exit 或者_Exit，又或者使用一个外部信号来使得该进程终止。

通常来说，在 Linux 中运行一个用户自行设计的可执行文件的流程可以简单表达，如图 2.13 所示。

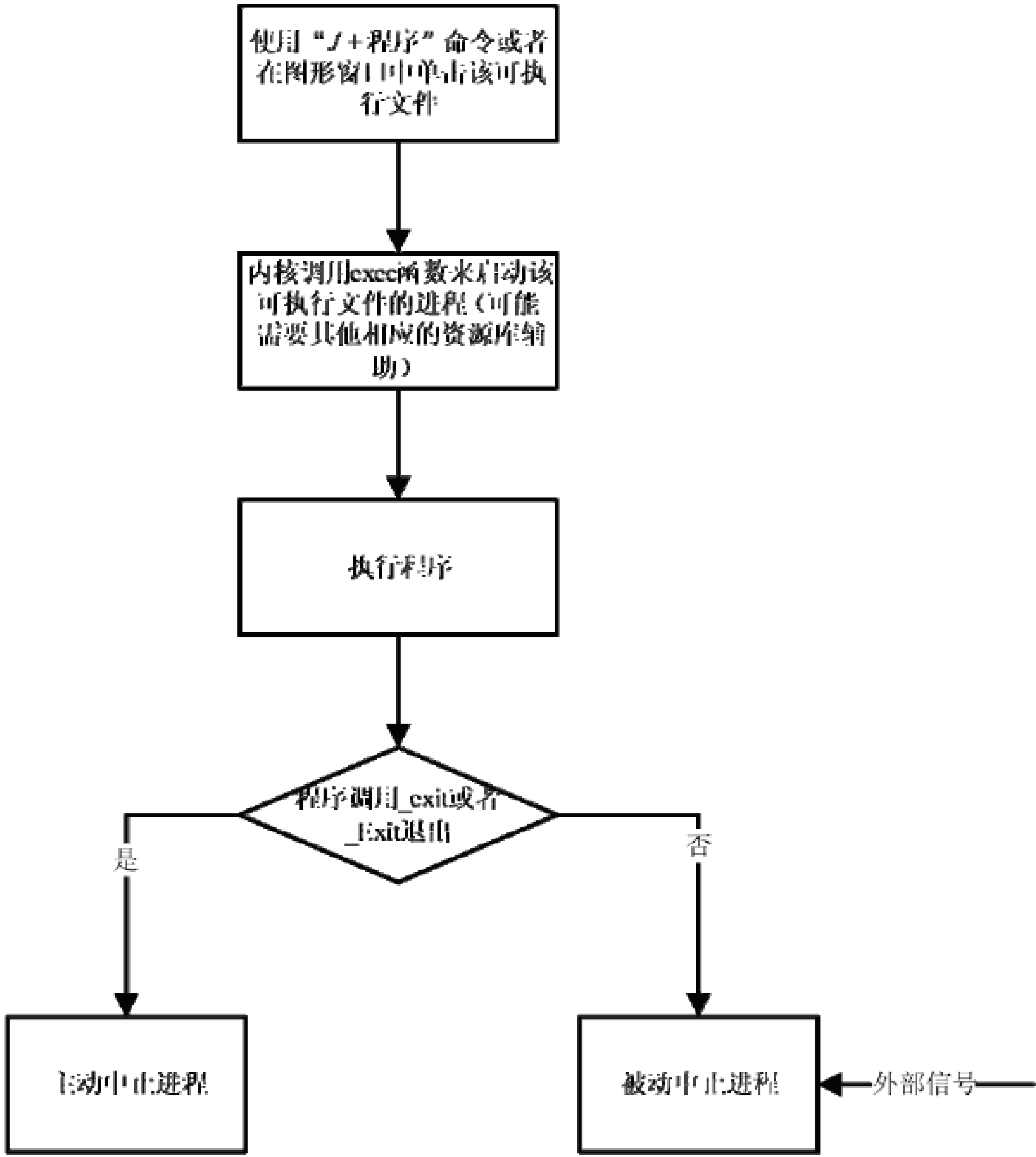


图 2.13　用户程序的运行过程

2.7.2　C 语言代码的程序存储空间

首先需要明确的是，本小节所讨论的程序空间是指用户的 C 语言代码编译生成的可执行文件，而不是 C 语言源代码。

这些可执行文件的存储空间可以分为如下几个部分。

- 正文段：存放了处理器执行的机器指令，通常来说正文段是可以共享的，所以包括 shell、gcc 在内的程序在存储器中只需要一个副本；正文段通常来说也是只读的，这是为了防止程序的可执行代码被意外修改。
- 初始化数据段：初始化数据段通常又被称为数据段，其包含了程序中需要进行初始化的变量值，例如如下的变量声明：

```
int counter = 0;
//counter 被初始化为 0，然后存放在初始化数据段中
//通常来说这些变量会是全局变量
//因为非全局变量会在调用时再分配空间并且进行初始化
```

- 非初始化数据段：非初始化数据段是和初始化数据段对应的，用来存放不需要初始化（其实是被自动初始化为 0 或者空指针）的变量，这个段又被称为 bss 段。
- 栈：这个段用于存放自动变量以及每次函数调用时需要保存的信息。
- 堆：用于动态存储分配，这个段位于非初始化数据段和栈之间，在很多场合下这个段和栈一起被合称为堆栈段。


注　意

对于一个可执行文件而言，其通常还有若干其他类型的段，例如包含了符号表的段，包含了 gdb 调试信息的段和包含了动态共享库链接表的段等，但是这些段并不会在进程调用的时候被装入存储区中去。

在 Shell 命令行，可以使用 size 命令查看一个可执行文件的正文段、数据段和 bss 段的长度信息，其单位是字节，对例 2.2 所生成的可执行文件使用 size 命令的执行过程如下。

```
alloy@ubuntu:~/linuxc/chapter2$ size exam201hello
   text    data     bss     dec     hex filename
   1183    504      16      1703    6a7 exam201hello
```

图 2.14 是 Linux 中对于这些段的典型安排方式，正文段通常从 0x0804800 地址单元开始，而栈底则位于 0xC0000000 之下从高地址向低地址方向增长。

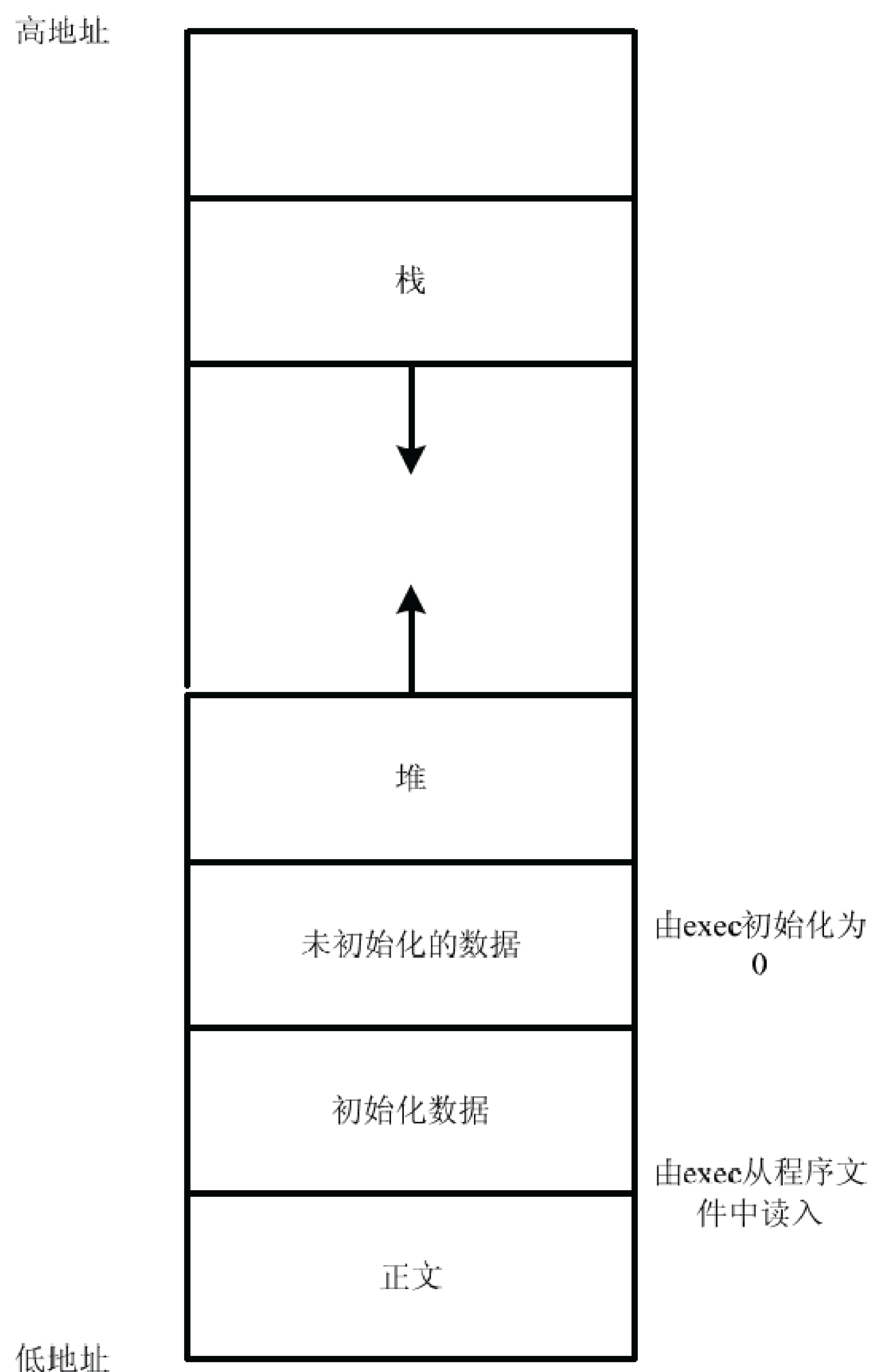


图 2.14　Linux 中典型的段分配方式

2.7.3　C 语言代码的 main 函数和参数

在 Linux 中，C 语言文件生成的可执行文件被 exec 调用，然后总是从 main 函数开始执行，第 1 章中所给出的 main 函数都是不带参数的，对在 Linux 下 main 函数的标准调用格式说明如下：

```
int　main(int argc, char *argv[])
```

在 main 函数的两个参数中，argc 必须是整型变量，其是命令行参数的数目；argv 必须是指向字符串的指针数组，这些指针分别指向各个命令行参数。

当 Linux 使用 exec 函数来启动一个 C 语言文件生成可执行文件的时候，其在调用 main 函数之前首先调用一个特殊的启动例程，并且将此启动例程指定为程序的起始位置，这个启动例程将从内核取得该可执行文件的命令行参数和环境变量值，然后传递给 main 函数。

当用户要运行一个可执行文件时，在 Linux 命令行下键入文件名，再输入实际参数即可把这些实参传送到 main 函数中去。

Linux 命令行的一般形式为：

```
可执行文件名　参数　参数……;
```

但是应该注意的是，main 的两个形参和命令行中的参数在位置上不是一一对应的，因为，main 的形参只有两个，而命令行中的参数个数原则上未加限制。argc 参数表示了命令行中参数的个数(注意：可执行文件名本身也算一个参数)，argc 的值是在输入命令行时由系统按实际参数的个数自动赋予的。例如有命令行为：

```
gcc hello.c -o hello
```

由于文件名 gcc 本身也算一个参数，所以共有 4 个参数，因此 argc 取得的值为 4。argv 参数是字符串指针数组，其各元素值为命令行中各字符串（参数均按字符串处理）的首地址。 指针数组的长度即为参数个数，数组元素初值由系统自动赋予。在上面的命令中，agv 数组的第 1 个元素指向的字符串为“gcc”，第 2 个元素指向的字符串为“hello.c”，第 3 个元素指向的字符串为“-o”，第 4 个元素指向的字符串为“hello”。


注 意

main 函数的参数可以省略掉在应用过程中的参数输入读取步骤，例如如果需要打开一个文件，可以直接在命令行中将文件名传递给应用代码，而不需要在应用代码中调用相应的输入代码等待用户输入；在实际使用中，如果不需要传递参数，也常常可以省略掉 main 函数的参数，直接写为 int main(void)。

此外，main 函数也带有返回值，默认的返回值类型为 int，在一般的程序中， main 函数的返回值类型 int 可以省略不写，返回值会直接传递给 Linux 内核，如果 main 函数的最后没有写 return 语句的话，gcc 会自动在生成的目标文件中加入“return 0;”，表示程序正常退出，main 函数的返回值可以将执行的相应结果反馈给内核，例如使用一个多判断语句分别返回不同的 int 值。

【例 2.10】main 函数的参数应用实例

这是一个 main 函数的参数应用实例，分别打印传递给 main 函数的参数数目以及参数内容。实例的应用代码如下：

```
#include <stdio.h>
int main(int argc,char *argv[])     //第一个存放参数的个数，第二个缓冲区存放参数
{
        unsigned int i=0;
        printf("%d\n",argc);
        for(i=0;i<argc;i++)
        {
                printf("%s\n",argv[i]);
        }
        return 0;
}
```

将文件保存为 exam203main.c，在终端中使用 gcc 编译，并且带命令行运行，可以看到执行过程如下。

```
alloy@ubuntu:~/linuxc/chapter2$ gcc exam203main.c -o exam203main          //编译命令
alloy@ubuntu:~/linuxc/chapter2$ ./exam203main this is a test!              //传递参数
```

```
5                                                    //参数数目
./exam203main                                        //以下输出字符串
this
is
a
test!
```

其中，第一行是 gcc 的编译命令行，其使用“-o”参数将 exam203main.c 文件编译成一个可执行文件 exam203main，然后执行，其中传递给 main 函数的参数字符串是“this is a test!”，这个参数字符串包括了 4 个字符串，然后算上可执行文件本身的字符串，所以 argc 的数目是 5。

2.7.4 C 语言代码的出错处理

在程序运行中，常常会出现各种错误，这些错误可能是在调用函数时由于逻辑问题产生的，也有可能是打开文件时发现文件不存在产生的，所以在进行相应的设计时必须要考虑到对错误的时间处理。在 Linux 中，如果调用的函数出现出错事件，往往会返回一个负值，并且此时整型变量 errno 常常会被设置为含有附加信息的一个值。

在 errno.h 文件中 Linux 定义了常用的错误常量，对这些错误说明如表 2.16 所示。

表 2.16 Linux 的常用错误常量

错误名称	说明
E2BIG	参数太长
EACCES	权限不够
EADDRINUSE	地址已经被使用
EADDRNOTAVAIL	无效的地址
EAFNOSUPPORT	被请求的地址不合法
EAGAIN	临时资源不够
EALREADY	连接已经建立
EBADE	非法的交换
EBADF	文件描述符不正确
EBADFD	文件描述符状态不正确
EBADMSG	消息出错
EBADR	请求描述符错误
EBADRQC	不正确的请求码
EBADSLT	插槽错误
EBUSY	资源/设备忙
ECANCELED	取消操作
ECHILD	没有子进程
ECHRNG	通道数溢出
ECOMM	通信中发送出错

（续表）

错误名称	说明
ECONNABORTED	连接中止
ECONNREFUSED	拒绝连接
ECONNRESET	连接复位
EDEADLK	避免资源死锁
EDEADLOCK	同步资源死锁
EDESTADDRREQ	缺少目的地址
EDOM	函数数学参数错误
EDQUOT	磁盘空间超限额
EEXIST	文件存在
EFAULT	地址错误
EFBIG	文件太大
EHOSTDOWN	服务器已经关闭
EHOSTUNREACH	不能连接到服务器
EIDRM	移除标识符
EILSEQ	非法的字节顺序
EINPROGRESS	操作进程已经存在
EINTR	中断服务字函数
EINVAL	参数不合法
EIO	输入/输出错误
EISCONN	socket 已经连接
EISDIR	这是一个目录
EISNAM	这是一个命名类型文件
EKEYEXPIRED	密钥已经过期
EKEYREJECTED	服务不允许使用这个密钥
EKEYREVOKED	密钥已经被停止使用
EL2HLT	2 级停机
EL2NSYNC	2 级未同步
EL3HLT	3 级停机
EL3RST	3 级重启
ELIBACC	不能连接需要的共享库
ELIBBAD	连接到一个不能使用的共享库
ELIBMAX	尝试连接多个共享库
ELIBSCN	库区域出错
ELIBEXEC	库路径错误

（续表）

错误名称	说明
ELOOP	符号连接级别太多
EMEDIUMTYPE	媒体类型错误
EMFILE	尝试打开多个文件
EMLINK	尝试多个链接
EMSGSIZE	消息过长
EMULTIHOP	多次尝试
ENAMETOOLONG	文件名过长
ENETDOWN	网络服务已经关闭
ENETRESET	网络连接已经中止
ENETUNREACH	没有网络连接
ENFILE	系统中打开的文件过多
ENOBUFS	没有足够的缓冲空间
ENODATA	流同步没有可获得消息源
ENODEV	没有这个设备
ENOENT	没有这个文件或者目录
ENOEXEC	格式错误
ENOKEY	不能获得需要的密钥
ENOLCK	不能获得必要的锁
ENOLINK	连接错误
ENOMEDIUM	没有媒体
ENOMEM	空间不够
ENOMSG	没有相应类型的消息
ENONET	计算机没有连接到网络
ENOPKG	包尚未安装
ENOPROTOOPT	不支持的协议
ENOSPC	没有多余的磁盘空间
ENOSR	流文件没有找到源
ENOSTR	不是流文件
ENOSYS	没有提供这个函数
ENOTBLK	需要一个块设备
ENOTCONN	socket 未连接
ENOTDIR	这不是一个目录
ENOTEMPTY	目录非空
ENOTSOCK	不是一个 socket

（续表）

错误名称	说明
ENOTSUP	不支持这样的操作
ENOTTY	不合法的 I/O 操作
ENOTUNIQ	网络名称错误
ENXIO	在对应地址没有找到设备
EOPNOTSUPP	不支持在 socket 上进行对应的操作
EOVERFLOW	变量的数据类型不支持这么大的数据
EPERM	不允许操作
EPFNOSUPPORT	不支持这样的协议簇
EPIPE	管道错误
EPROTO	协议错误
EPROTONOSUPPORT	不支持这样的协议
EPROTOTYPE	socket 协议类型错误
ERANGE	结果太大
EREMCHG	远程地址已经修改
EREMOTE	目标不在本机
EREMOTEIO	远程 I/O 错误
ERESTART	中断服务系统需要重启
EROFS	系统文件只读
ESHUTDOWN	传输端点已经关闭，不能发送
ESPIPE	不合法的查找
ESOCKTNOSUPPORT	不支持这种 socket 类型
ESRCH	没有这种进程
ESTALE	文件句柄过期
ESTRPIPE	流管道错误
ETIME	计时器耗尽
ETIMEDOUT	连接超时
ETXTBSY	文本文件忙
EUCLEAN	结构需要被清除
EUNATCH	找不到协议驱动
EUSERS	用户太多
EWOULDBLOCK	操作被阻塞
EXDEV	不恰当的链接
EXFULL	交换空间满

在 Linux 系统中，errno 的定义如下，其可以是一个包含出错编号的整数，也可以是一个返回

出错编号指针的函数：

```
extern int errno;
```

对于 errno 而言，其有如下两条规则：

- 如果没有出错，则 errno 的值并不会被一个例程清除，因此可以当函数返回值指明出错的时候才去检查 errno 的值。
- 任何一个函数都不会把 errno 设置为 0。

Linux 使用 strerror 和 perror 函数来打印相应的出错信息，对这两个函数的标准调用格式说明如下：

```
#include <string.h>
char *strerror(int errnum);
#include <sdtio.h>
void perror(const char *msg);
```

strerror 函数的返回值是一个指向消息字符串的指针，这个消息字符串即为出错信息的字符串；而 peeror 函数没有返回值，其输出如下：

```
"由 msg 指针指向的字符串" + "：" + " " + "回车换行"
```

【例 2.11】strerror 和 peeror 函数应用实例

这是 strerror 函数和 peeror 函数的应用实例，其分别调用 strerror 函数输出了一个 EACCES 错误值对应的错误提示，调用 perror 函数给出了一个带调用执行文件名的错误提示。

实例的应用代码如下：

```
#include <string.h>
#include <stdio.h>
#include <errno.h>
int main(int argc, char *argv[])
{
        fprintf(stderr, "EACCES: %s\n", strerror(EACCES));
        errno = ENOSPC;     //传递错误标号
        perror(argv[0]);     //打印出错应用代码
        return 0;
}
```

将文件保存为 exam204error.c，在终端中使用 gcc 编译并且运行，可以看到如下的执行过程。

```
alloy@ubuntu:~/linuxc/chapter2$ gcc exam204error.c -o exam204error
alloy@ubuntu:~/linuxc/chapter2$ ./exam204error
EACCES: Permission denied
./exam204error: No space left on device
```

其中，第 1 行是 gcc 的编译命令行，其使用"-o"参数将 exam204error.c 文件编译成一个可执行文件 exam204error，然后执行，输出的第 3 行给出的是 EACCES 这个错误提示符给出的错误提

示内容，而第 4 行则是给出了产生错误的可执行文件名称“./ exam204error”和对应 ENOSPC 的错误提示。

2.7.5 C 语言代码的标准输入和输出函数

在实际应用中，Linux 的 C 语言代码需要和用户进行通信，此时可以使用 printf 函数和 scanf 函数，它们被称为标准输入输出函数。

1. 标准输出函数 printf

printf 函数用于将格式化数据输出，其标准调用格式如下：

```
#include <stdio.h>
int printf(const char *format, ...);
```

其参数 format 是一个字符串，包含字符、字符序列和格式说明，其中字符部分与字符序列按顺序输出；而格式说明以“%”开始，格式说明使跟随的相同序号数据按格式说明转换和输出。如果数据的数量多于格式说明，多余的数据将被忽略，如果格式说明多于数据，结果将是随机的。如果输出成功，函数的返回值为输出的字符数目，如果输出失败，则返回一个负数。

printf 函数的格式说明结构为：%_flags_width_.precision_{b|B|l|L}_type，对各个部分的说明如下。

- type 用来说明参数是字符、字符串、数字或者指针字符，如表 2.17 所示。

表 2.17 printf 函数的 type 参数

type	输出结果
D	有符号十进制数
U	无符号十进制数
O	无符号八进制数
x	无符号十六进制数，使用小写
X	无符号十六进制数，使用大写
f	格式为[-]ddd.ddd 的浮点数
e	格式为[-]d.ddde+dd 的浮点数
E	格式为[-]d.dddE+dd 的浮点数
g	使用 f 或者 e 中比较合适形式的浮点数
G	使用 f 或者 E 中比较合适形式的双精度值
c	单字符常数
s	字符串常数
p	指针，格式 t:aaaa，其中 aaaa 为十六进制的地址，t 为存储类型
n	无输出，但是在下一参数所指整数中写入字符串
%	“%”字符

- b、B、l、L 用于 type 之前，说明整型 d、i、u、o、x、X 的 char 或者 long 转换。

- flags 是标记，其用法如表 2.18 所示。

表 2.18 printf 函数的 flags 参数

flags	作用
-	左对齐
+	有符号，数值总是以正负号开始
空格	数字总是以符号或者空格开始
*	忽略

- width 是域宽，只能是一个非负数，用来表示输出字符的最小个数，如果打印字符较少则使用空格填充，在前面加负号则表示为在域中使用左对齐，加 0 则表示用 0 填充。如输出的字符个数大于域的宽度，仍然会输出全部的字符。“*”表示后续整数参数提供域的宽度，前面加 b，表示后续参数是无符字符。
- precision 精度，对于不同类型的意义不同，可能引起截尾或者舍入，如表 2.19 所示。

表 2.19 printf 函数的 precision 精度

数据类型	说明
d、u、o、x、X	输出数字的最小位，输出数字超出也不截断尾，如果超出在左边，则填入 0
f、e、E	输出数字的小数位数，末位四舍五入
g、G	输出数字的有效位数
c、p	无影响
s	输出字符的最大字符数，超过部分将不显示

2. 标准输入函数 scanf

和 printf 函数相对，标准输入函数 scanf 用于用户向程序输入数据，其标注调用格式如下：

```
#include <stdio.h>
int scanf(const char *format, ...);
```

其参数结构和 printf 完全相同，因此可以参考上一小节，如果函数调用成功则返回指定的输入项数，若输入出错，或在任意变换前已至文件尾端则为 EOF。

3. 标准输入输出函数的应用实例

【例 2.12】标准输入输出函数应用实例

这是 printf 和 scanf 函数的应用实例，其要求提示用户输入 a、b 两个整数，然后输出相乘的结果，再要求输出一个字符串后输出刚刚输入的字符串。

实例的应用代码如下：

```
#include <stdio.h>
int main(void)                                      //没有参数
{
   int a,b,sum;
```

```
    char str[30]; //字符串存放
    printf("please input a,b!\n");
    scanf("%d%d",&a,&b);              //输入两个整数
    sum = a * b;                      //计算乘积
    printf("the sum is %d\n",sum);    //输出计算结果
    printf("please input the string\n");
    scanf("%s",str);
    printf("the string is %s\n",str); //打印刚刚输入的字符串
    return 0;
}
```

将文件保存为 exam205IO.c，在终端中使用 gcc 编译并且运行，可以看到如下的执行过程。

```
alloy@ubuntu:~/linuxc/chapter2$ gcc exam205IO.c -o exam205IO
alloy@ubuntu:~/linuxc/chapter2$ ./exam205IO
please input a,b!
33
21
the sum is 693
please input the string
alloy
the string is alloy
```

其中，第一行是 gcc 的编译命令行，其使用“-o”参数将 exam3IO.c 文件编译成一个可执行文件 examIO，然后执行，分别输入需要输出的数据并且输出。



注 意

printf 和 scanf 其实都是 Linux 下的标准流操作函数，用于和用户的交互，关于它们的进一步说明可以参考第 5 章。

2.7.6 C 语言代码的时间处理

在 Linux 系统应用中，经常需要获得当前的时间信息，Linux 内核提供了一些相应函数用于操作，对其标准调用格式说明如下：

```
#include <time.h>
char *asctime(const struct tm *tm);
char *asctime_r(const struct tm *tm, char *buf);
char *ctime(const time_t *timep);
char *ctime_r(const time_t *timep, char *buf);
struct tm *gmtime(const time_t *timep);
struct tm *gmtime_r(const time_t *timep, struct tm *result);
struct tm *localtime(const time_t *timep);
struct tm *localtime_r(const time_t *timep, struct tm *result);
time_t mktime(struct tm *tm);
int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

各个时间函数的关系说明如图 2.15 所示，对其详细说明如下。

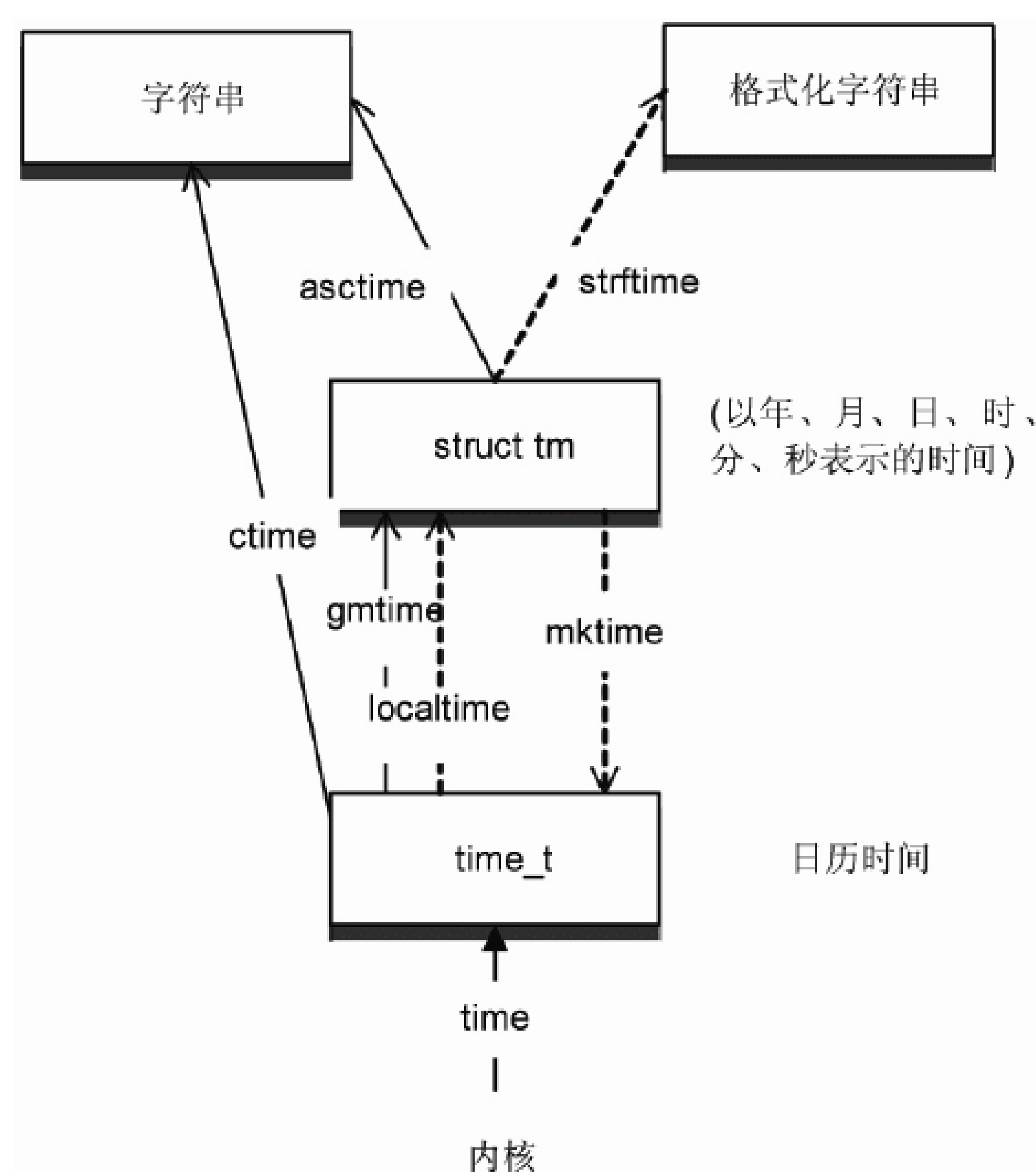


图 2.15　Linux 的时间函数的相互关系

- asctime 函数：将参数 timeptr 所指的 tm 结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果以字符串形态返回。此函数已经由时区转换成当地时间，字符串格式为："Wed Jun 30 21:49:08 1993/n"。该函数的参数值是 tm 指针指向的存储空间，返回值是表示目前当地时间日期的字符串。



注　意

若再调用相关的时间日期函数，此字符串可能会被破坏。此函数与 ctime 的不同之处在于传入的参数是不同的结构。

对 asctime 函数中涉及的 tm 时间信息结构体说明如下。

```
struct tm
{
   int tm_sec;                  //秒
   int tm_min;                  //分钟
   int tm_hour;                 //小时
   int tm_mday;                 //日期
   int tm_mon;                  //月份
   int tm_year;                 //年份
   int tm_wday;                 //星期
   int tm_yday;                 //从 1 月 1 日开始到当日日期编号
   int tm_isdst;
  //夏令时标识符，实行夏令时的时候，tm_isdst 为正，不实行夏令时的时候，
  //tm_isdst 为 0，不了解情况时，tm_isdst()为负
};
```

- asctime_r 函数：是 asctime 函数的一个扩展，提供了一个缓冲器件 buf 用于存放返回值，该缓冲区的长度不能低于 26 个字节。
- ctime 函数：将参数 timep 所指的 time_t 结构中的信息转换成真实世界所使用的时间日期

表示方法，然后将结果以字符串形态返回。此函数已经由时区转换成当地时间，字符串格式为 “Wed Jun 30 21 :49 :08 1993/n”。若再调用相关的时间日期函数，此字符串可能会被破坏。

- char *ctime_r 函数：和 ctime 函数功能相同，也是提供了一个缓冲区用于存放返回值。
- gmtime 函数：将所指的 time_t 结构中的信息转换为真实世界所使用的时间日期表示方法，然后将结果返回到 tm 结构体中。
- gmtime_r 函数：和 gmtime 函数类似，同时提供了一个由 result 指针指向的内存空间，用于存放返回值。
- localtime 函数：当地的目前时间和日期，其将参数 timep 所指的结构体中的信息转换为真实世界所使用的时间日期表示方法，然后返回。
- localtime_r 函数：和 localtime 函数类似，同时提供了一个由 result 指针指向的内存空间，用于存放返回值。
- mktime 函数：将参数 tm 所指向的结构体数据转换为从 1970 年 1 月 1 日 0 时 0 分 0 秒开始所经历的秒数，然后返回。
- gettimeofday 获取当前时间和时区信息，这个需要超级用户的权限，tv 参数用于指向存放返回时间信息的缓冲区，对其结构说明如下。

```
struct timeval {
      time_t          tv_sec;       // 秒
      suseconds_t tv_usec;       // 微妙
};
```



注　意

而 tz 用于存放相应的时钟信息，说明如下：

```
struct timezone {
      int tz_minuteswest;         //minutes west of Greenwich
      int tz_dsttime;             //type of DST correction
};
```

- settimeofday 用于设置当前时间和时区信息，其参数和使用方法可以参考 gettimeofday。

【例 2.13】打印当前 Linux 系统时间信息

这是一个系统时间函数的应用实例，其调用了 time、localtime 和 asctime 函数用于在屏幕上打印当前的 Linux 系统时间信息。应用代码首先调用 gmtime 函数获得当前时间的秒数据，然后使用 asctime 函数将该秒数据转换为正常的显示格式。

实例的应用代码如下：

```
//打印系统的当前时钟
#include <time.h>
#include <stdio.h>
int main(void)
{
          time_t timetemp;                              //定义一个时间结构体变量
          char *wday[] = {"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
```

```
        struct tm *p;                                //结构体指针
        time(&timetemp);                             //获得时间参数
        printf("%s",asctime(gmtime(&timetemp)));
        p = localtime(&timetemp);
        printf("%d:%d:%d:\n",(1900+p->tm_year),(1+p->tm_mon),p->tm_mday);
        printf("%s    %d:%d:%d\n",wday[p->tm_wday],p->tm_hour,p->tm_min,p->tm_sec);
        return 0;
}
```

将文件保存为 exam206time.c，在终端中使用 gcc 编译并且运行，可以看到如下的执行过程：

```
alloy@ubuntu:~/linuxc/chapter2$ gcc exam206time.c -o exam206time
alloy@ubuntu:~/linuxc/chapter2$ ./exam206time
Sat Aug 23 01:56:11 2014
2014:8:23:
Sat    9:56:11
```

其中第 1 行是应用代码的编译过程，第 2 行是调用执行，第 3~5 行是输出的时间信息。

【例 2.14】计算代码运行时间

这是一个使用时间函数来测试当前程序所需运行时间的实例，应用代码首先调用 gettimeofday 获得当前的时间信息，然后将这些时间信息分门别类地输出，最后测试了本身的执行时间。

实例的应用代码如下：

```
//获得秒和微秒时间，现实和 Greenwich 的时间差，并且测试运行这段程序所需要的时间
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>
int main(void)
{
        struct timeval time1,time2;
        struct timezone timez;
        gettimeofday(&time1,&timez);                 //获得当前的时间
        printf("tv_sec; %d\n",time1.tv_sec);         //秒
        printf("tv_usec; %d\n",time1.tv_usec);       //毫秒
        printf("tz_minuteswest; %d\n",timez.tz_minuteswest);
        printf("tz_dsttime; %d\n",timez.tz_dsttime);
        gettimeofday(&time2,&timez);
        printf("time2_usec-time1_usec; %d\n",(time1.tv_usec - time2.tv_usec));
        //计算程序执行的时间
        return 0;
}
```

将文件保存为 exam207gettime.c，在终端中使用 gcc 编译，并且带命令行运行，可以看到如下的执行过程，其中的警告部分内容是由于变量类型不匹配导致的，可以不予理睬，或者进行相应的类型转换操作即可。

```
alloy@ubuntu:~/linuxc/chapter2$ gcc exam207gettime.c -o exam207gettime
……//省略了部分警告信息
alloy@ubuntu:~/linuxc/chapter2$ ./exam207gettime
tv_sec; 1408759081
```

```
tv_usec; 809745
tz_minuteswest; -480
tz_dsttime; 0
time2_usec-time1_usec; -41
```

其中第一行为应用代码的编译过程，由于变量类型不匹配出现了一些警告，但不影响使用，所以直接忽略，执行后可以看到对应的执行时间输出。

2.7.7 C 语言代码的分配机制

Linux 操作系统提供了三个用于存储空间动态分配的函数和一个用于释放内存空间的函数，对这 4 个函数详细说明如下。

- malloc 函数：给进程分配指定字节数的存储区，此存储中的初始值不为 0。
- calloc 函数：为指定流数量具有指定长度的对象分配存储空间，该空间中每一位都被初始化为 0。
- realloc 函数：更改以前分配区的长度（可以增加，也可以减少），当为增加长度时，可能需要将以前分配区间的内容迁移到另外一个足够大的区域，在尾部提供增加的存储区，而新增加的区间内的初始化值不确定。
- free 函数：用于释放其参数指针指向的存储空间，这些空间会被送入系统的可用存储区池，可以被以上三个函数再次分配。

对这 4 个函数的标准调用格式说明如下，三个分配函数如果调用成功，则返回一个指向分配区的非空指针，否则返回空指针，而 free 函数没有返回值。

```
#include    <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj,size_t size);
void *realloc(void *ptr,size_t newsize);
void free(void *ptr);
```

内存分配函数所返回的指针一定是适当对齐的，从而使得这些存储空间可以应用于任何数据对象，并且由于其返回值均为通用指针 void*，当用户使用它们的时候，通常是不需要进行类型转换的。

在这三个内存分配函数中，realloc 函数使得用户可以增加或者减少以前分配的内存空间的长度，例如可以使其减少使用固定长度的数组，从而节省了内存空间，但是需要注意的是最后一个参数 newsize 是新分配的存储区长度，而不是分配后存储区的总长度，如果 ptr 指向一个空指针的话，则 realloc 函数的功能和 malloc 是完全相同的。

另外这三个函数通常都是通过调用 sbrk 系统调用来实现的，对该系统调用的标准化格式说明如下：

```
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t increment);
```



注 意

在内存空间使用完成之后必须立即释放，否则可能导致内存泄漏，这是 Linux 系统开发中最常见的问题之一。如果以前分配的一片内存不再需要使用或无法访问时，但是却并没有释放它，那么对于该进程来说，会因此导致总可用内存的减少，这时就出现了内存泄漏。

2.7.8　C 语言代码的系统调用和库函数

Linux 内核提供了一些内建的函数可以用来完成一些系统级别的功能，这样的函数称为“系统调用”，英文是 systemcall，这些函数代表了从用户空间到内核空间的一种转换。

系统调用的相关声明可以在 syscall.h 中找到，这些系统调用都对应一个具体的数字，Liunx 内核通过位于 0x80 中断来管理这些系统调用，而这些系统调用的对应数字和相应参数都在被调用的时候送到对应寄存器里。



注 意

系统调用的数字实际上是一个序列号，表示其在系统的一个数组 sys_call_table[]中的位置。

在具体的使用中，Linux 为这些系统调用在标准 C 函数库中设置了一个具有相同名字的函数，用户可以通过相应的调用方法来对这些函数进行调用，然后该函数使用系统所需要的技术调用相应的内核服务，所以从应用角度来说，可以将这些系统调用看成 C 语言函数。

另外 Linux 还提供了一些通用库函数，以供用户调用，但是虽然这些函数可以调用一个或者多个内核的系统调用，但是它们并不是内核的入口点，例如 atoi 函数等。

从操作系统的角度来看，系统调用和库函数的实现方法有重大的区别，但是从用户（Linux 下 C 程序员）的角度来看它们是一样的，在很多实际应用中应用程序会调用系统调用或者库函数，而库函数又会调用系统调用，它们的关系如图 2.16 所示。

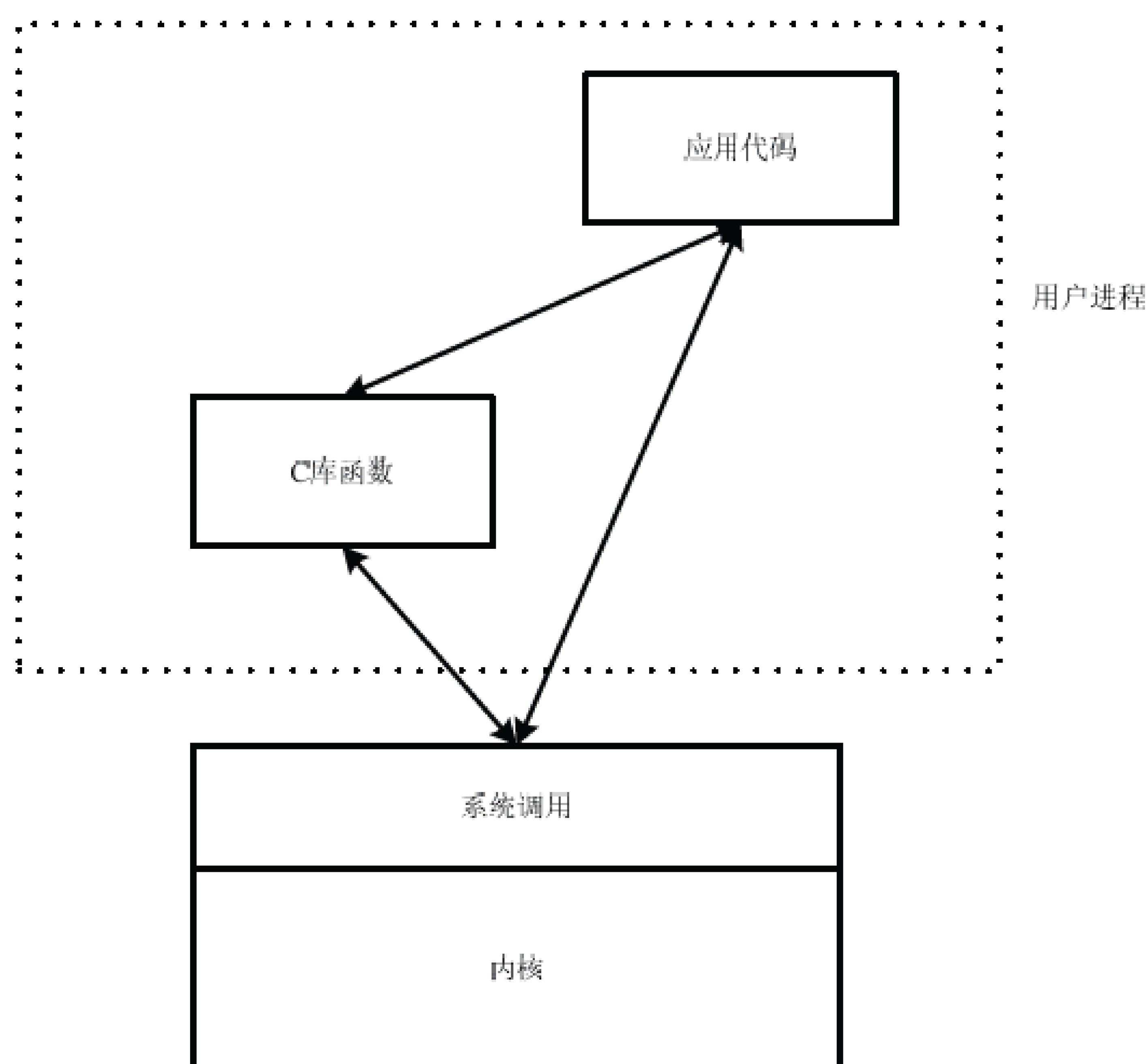


图 2.16　Linux 下库函数和系统调用的关系


注　意

系统调用通常只提供一种最小的接口，而库函数通常会提供比较复杂的功能。在必要的时候用户可以自行替换或者修改库函数，但是不能替换或者修改系统调用。

2.8　本章习题

1. 简述 Linux 下的 C 语言开发流程以及每个步骤对应的工具。

2. 参考第 2.3.1 小节，熟悉 vim 编辑环境的使用方法。

3. 参考第 2.4 节对 gcc 进行安装配置，并且参考例 2.2 和例 2.3 完成对 C 语言文件的编译、链接和执行。

4. 参考第 2.5 节学习 gdb 的使用方法。

5. 参考例 2.10 来熟悉 main 函数的 argc 和 argv 参数的使用方法。

6. 参考表 2.16 中的错误名称测试不同的 Linux 错误状态。

7. sqrtf 是平方根函数，对其标准调用格式说明如下（可以使用 man 命令获得更多的信息）：

```
#include <math.h>
float sqrtf(float x);
```

使用该函数以及 scanf 和 printf 函数实现从键盘输入 n 个实型数据，分别求其对应的平方根，并且在屏幕上输出。

8. 使用 malloc 函数编写一段程序，用于模拟在内存中为一个手机的通讯录增加存储空间的情况，该通讯录的结构体定义为 struct co，对其中各个分量说明如下。

- index：编号。
- name：姓名。
- MTel：手机号码。
- Tel：座机号码。

第 3 章　Linux 文件的基础操作

Linux 中的文件是指以计算机的存储设备为载体的信息集合，简单地说就是保存在硬盘、内存、光盘等设备上的一个个由各种数据组合在一起的实体，是 Linux 在物理上最基本的组成单元。本章将介绍与 Linux 文件相关的基础知识，以及如何使用 C 语言对文件进行相应的操作，涉及以下内容：

- Linux 下的文件类型和结构。
- 在 Linux 下创建文件、打开文件和关闭文件。
- 向文件写入数据和读出数据。

3.1　Linux 的文件

Linux 中有许多文件，这些文件可能是在安装 Linux 时系统自带的文件，也可能是用户安装的各种应用软件，还可能是用户自己创建的文件。在终端中使用 ls-l 命令可以看到根目录下的如下文件列表：

```
alloy@ubuntu:/$ ls -l
总用量 88
drwxr-xr-x    2 root root   4096   1 月  29 21:53 bin
drwxr-xr-x    3 root root   4096   2 月   7 11:07 boot
drwxr-xr-x    2 root root   4096   1 月  29 21:04 cdrom
……                              //其间省略了部分内容
lrwxrwxrwx     1 root root      33    2 月    7 11:06 vmlinuz -> boot/vmlinuz-3.2.0-37-generic-pae
lrwxrwxrwx     1 root root      33    1 月   29 21:53 vmlinuz.old -> boot/vmlinuz-3.2.0-36-generic-pae
```

在图形界面（Unity）中打开根目录会发现这些文件的表现形式，如图 3.1 所示，和终端中的列表对比可以发现它们是一一对应的。

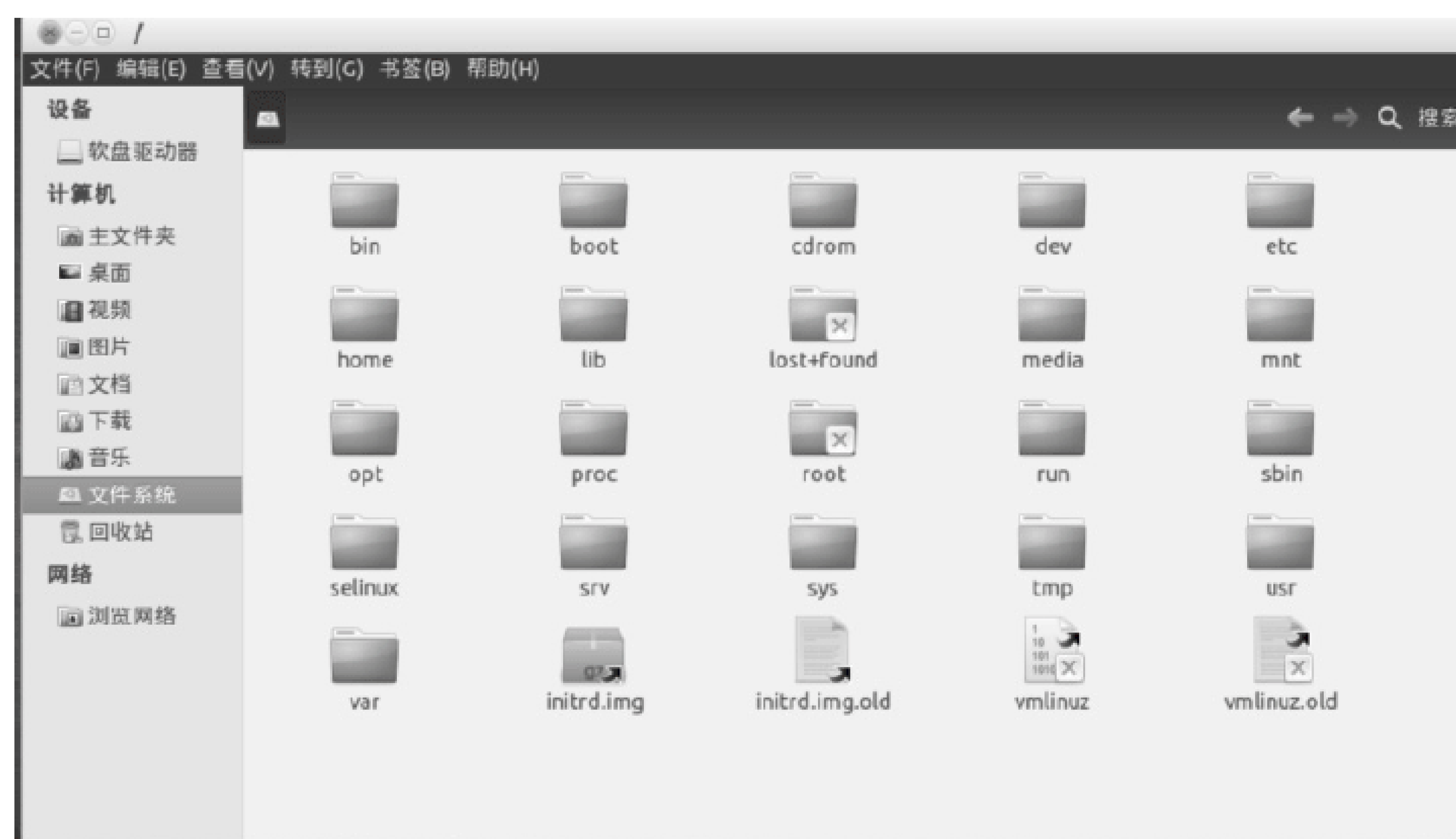


图 3.1　图形界面中根目录下的文件

3.1.1 Linux 的文件类型

Linux 的文件通常可以分为 7 大类（7 种文件类型），如表 3.1 所示。

表 3.1 Linux 中的文件类型

文件类型	缩写	英文名称
普通文件	-	Regular file
目录文件	D	Directory file
块设备文件	B	Block special file
字符设备文件	C	Character special file
命名管道文件	F	FIFO 或者 named pipe
套接字文件	S	Socket
符号链接文件	l	Symbolic link

1. 普通文件

普通文件是 Linux 系统中最常见的一类文件，其特点是不包含文件系统的结构信息，可包括图形文件、数据文件、文档文件、声音文件等；普通文件按其内部结构又可分为文本文件和二进制文件两种。

（1）文本文件

文本文件是字符（ASCII 码）组成的文件，以行为基本结构的信息存储文件，其是 Linux 系统中最多的一种文件类型，它的内容是用户可以直接读到的数据，例如数字、字母等。通常来说，Linux 的系统配置文件基本上都属于这种文件类型，可以使用 cat 命令直接查看。

（2）二进制文件

二进制文件是按信息在内存中的格式表示的文件，其通常不能直接查看，而必须使用相应的软件来查看。通常来说，Linux 中的可执行文件（脚本、文本方式的批处理文件不算）基本都属于这种文件类型，可以运行。

2. 目录文件

Linux 中的目录也是以文件存在的，称为目录文件，其是文件系统中一个目录所包含的目录项组成的文件，用户可以读取但是不能修改该目录文件的内容，只允许系统进行修改。

目录文件在文件名与索引节点之间的转换起到桥梁作用，是文件系统树形文件结构的关键，其由文件名和索引节点号构成。

Linux 的文件系统对文件的管理是通过索引节点来进行的，目录文件只不过提供了文件名和索引节点之间的转换手段。为了保证文件系统层次的完整性，目录文件是由系统来管理的，用户只能读目录文件，而不允许直接写目录文件。每个目录文件的前 2 项是 2 个特设的文件“.”和“..”，其中“.”对应于该目录文件本身的索引节点，而“..”则对应于其父目录的索引节点。如果一个目录中只包含“.”和“..”文件，则该目录为空目录。

当用户访问某个文件时，系统需要找到它所对应的索引节点。目录文件建立了文件名和索引

节点号之间路径的路线，例 3.1 是对目录 a 下名称为 b 的文件访问流程。

【例 3.1】文件访问流程

文件 b 的路径存在形式是“../a/b”，要从当前目录开始，到达其父目录，再到达其父目录的子目录 a，然后访问文件 b，其详细操作步骤如下：

01 检索当前目录的索引节点。
02 通过当前目录的索引节点，找到当前文件，查出父目录“..”。
03 检索“..”的索引节点。
04 通过父目录“..”的索引节点，检索父目录文件，查出文件“a”的索引节点号。
05 检索“a”的索引节点。
06 利用“a”的目录索引节点中的信息，检索“a”目录文件，查到“b”的索引节点号。
07 检索文件“b”的索引号。
08 访问文件“b”。

由于在系统的内存中存在内存索引节点表，所以以上的操作速度会很快。

3. 字符设备文件和块设备文件

Linux 把设备（例如硬盘、串口等）也看做文件，具有相同的操作方法，这种文件被称为设备文件，其是用于操作系统与 I/O 设备提供连接的一种文件，分为字符设备文件和块设备文件，分别对应于字符设备和块设备，这些文件通常存放在 dev 目录中。


注 意

Linux 中存在一个目录 /dev/null，所有放入这一设备的数据都将不存在，可以把此放入操作看成是删除。

- 字符设备文件：这是一个顺序的数据流设备文件，对这种文件的读写是按字符进行的，而且这些字符是连续地形成一个数据流。字符设备不具备缓冲区，所以对这种设备的读写是实时的，如串口终端、磁带机等。
- 块设备文件：这是一种具有一定结构的随机存取设备文件，对这种设备的读写是按块进行的，它使用缓冲区来存放暂时的数据，待条件成熟后，从缓存一次性写入设备，或从设备中一次性读出放入到缓冲区，如磁盘和文件系统等。

【例 3.2】串口和硬盘对应的设备文件

例 3.2 是使用 ls -l 命令来查看串口和硬盘对应设备文件（分别为/dev/tty 和/dev/had1）的信息，可以看到/dev/tty 的文件类型显示为字符“c”，/dev/hda1 的文件类型显示为字符“b”。

```
# ls -l /dev/tty
crw-rw-rw- 1 root tty 5,    0 04-19 08:29    /dev/tty
# ls -l /dev/hda1
brw-r-----    1 root disk 3, 1 2006-04-19    /dev/hda1
```

4. 命名管道文件

命名管道文件又被称为先进先出文件，其主要用于在 Linux 的进程间传递数据，其是 Linux 进程间的一种通信机制。管道是进程间传递数据的“媒介”，一个进程将数据写入管道的一端，另一个进程从管道另一端读取数据。通常情况下，管道是建立在高速缓存中的。采用先进先出的规定处理其中的数据，管道文件又可以分为有名管道和无名管道两种，本书将在后面的章节中进行详细介绍。

5. 套接字文件

套接字（Socket）文件主要用于在不同计算机进程间的通信，其是操作系统内核中的一个数据结构，它是网络中的节点进行相互通信的门户。套接字有三种类型：流式套接字、数据报套接字和原始套接字。流式套接字也就是 TCP 套接字（或称面向连接的套接字），数据报套接字也就是 UDP 套接字（或称无连接的套接字），原始套接字用“SOCK_RAW”表示。

- 流式套接字定义了一种可靠的面向连接服务，实现了无差错、无重复的顺序数据传输。
- 数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证可靠、无差错。
- 原始套接字允许对低层协议，如 IP 或 ICMP 进行直接访问，主要用于对新的网络协议进行测试等。

6. 符号链接文件

符号链接文件又称链接文件，其是一种特殊的文件，实际上是指向一个真实存在的文件链接。链接文件提供了共享文件的一种方法，在链接文件中不是通过文件名实现文件共享，而是通过链接文件所包含的指向文件的指针来实现对文件的访问。普通用户可以建立链接文件，并通过其指针访问它所指向的那个文件。使用链接文件可以访问普通文件，还可以访问目录文件和不具有普通文件形态的其他文件，也就是说，链接文件可以在不同的文件系统之间建立一种链接关系。根据链接对象的不同，链接文件又可以分为硬链接文件和符号链接文件。

3.1.2　Linux 的文件结构和文件描述符

Linux 的文件是个简单的字节数据序列，所以在 Linux 下对于文本文件、二进制文件的结构和访问方法都是相同的。Linux 的文件是由一系列块（block）组成，每个块可能含有 512、1024、2048 或 4096 个字节，具体由系统实现决定，在同一个文件系统中块大小是相同的。当使用较大块的时候，由于每次磁盘操作可以传输更多的数据，操作所花的时间较少，所以可以提高磁盘和内存间数据的传输率，但是相对的，由于块太大，存储的有效容量也会下降，也就是说会浪费一些存储空间。

Linux 使用文件描述符（File Descriptor）来标识一个进程正在访问的特定文件，当打开一个文件或者创建一个文件时，Linux 将返回一个文件描述符，以供其他操作引用，通常来说文件描述符是一个小的非负整数。


注　意

文件描述符是对应进程的，每一个文件描述符都对应一个特定的文件，而每一个特定的文件可以对应不同的进程存在多个不同的文件描述符。

在 Linux 中，每个进程都可以拥有最多 1024 个文件描述符，并且有自己的文件描述符表，其中前三项对于一般的进程是固定的，且是由系统自动打开的，说明如下。

- 文件描述符 0：标准输入文件，通常对应键盘等输入设备。
- 文件描述符 1：标准输出文件，通常对应显示设备。
- 文件描述符 2：标准错误输出文件，通常也是对应显示设备。

对于以上三个文件描述符，用户程序不用执行文件打开操作就可直接使用，其在头文件中的定义部分如下：

```
#define    STDIN_FILENO     0    //标准输入
#define    STDOUT_FILENO    1    //标准输出
#define    STDERR_FILENO    2    //标准错误输出
```

3.2 Linux 的文件基础操作

Linux 的文件基础操作包括打开文件、关闭文件、创建文件、向文件写入数据、从文件读出数据等，Linux 通过调用相应的文件 I/O 函数来完成相应的操作，这些函数包括 open（打开）、create（创建）、write（写）等。


注 意

这些函数通常被称为“不带缓冲的 I/O 操作函数”，其是对应流文件操作函数而言的，关于流文件操作函数的相应知识将在第 6 章进行介绍。

图 3.2 是一个常见的 Linux 中的文件操作流程，对于所有的 Linux 文件而言，在对其进行其他操作之前都必须首先打开文件，在进行操作完成之后必须要关闭文件；在进行读写操作的时候必须对读写的位置进行定位。

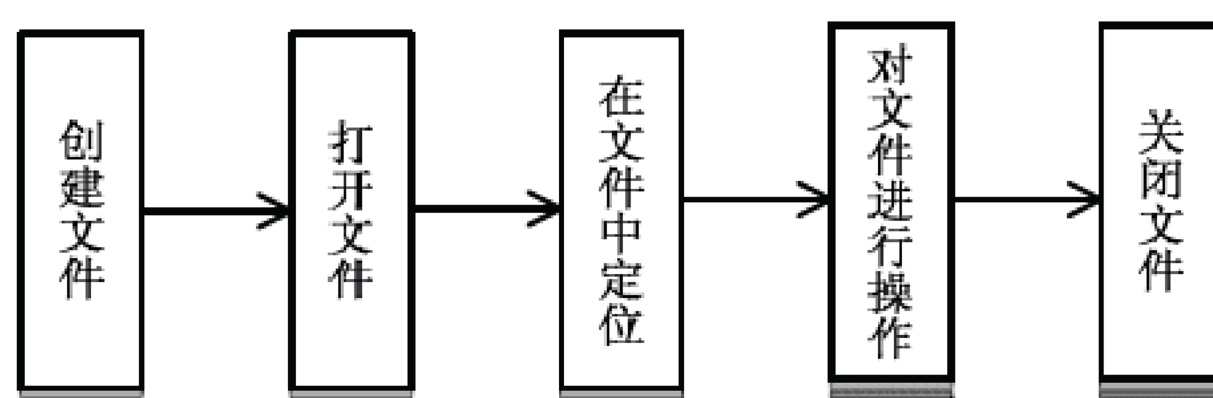


图 3.2　常见的文件操作流程


注 意

任何被打开的文件在操作完成之后必须关闭，否则容易出现错误。

3.2.1　打开和关闭文件

在 Linux 中如果想要打开一个文件可以调用 open 函数，该函数用于在 Linux 中打开一个文件，如果该文件不存在，则先创建该文件，然后打开，如果操作成功，则返回文件对应的文件描述符，如果操作失败，则返回“-1”。

对 open 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *pathname, int flags ) ;                //打开一个现有的文件
int open (const char *pathname, int flags, mode_t mode ) ;
//如果打开的文件不存在，则先创建它
```

对 open 函数的各个参数和应用实例说明如下。

1. open 函数的 pathname 参数说明

pathname 是一个指针变量，用于传递包含了路径的完整文件名称，其典型的应用实例是“/dev/log”。

2. open 函数的 flags 参数说明

flags 是一个 int 类型的变量，用于指定文件的打开方式，常用的标志有如下三种。

- 只读：关键字 O_RDONLY，通常定义为 0。
- 只写：关键字 O_WRONLY，通常定义为 1。
- 读写：关键字 O_RDWR，通常定义为 2。

在对一个文件进行相应的操作时，还必须注意文件本身的权限，对一个文件进行操作权限不够的操作将会返回一个错误，例如对只读文件进行写操作。

需要注意的是 flags 参数中以上三个参数是必须且唯一的，也就是说这些关键字之间不能用“OR”来连接，只能选择其中一个，此外 flags 还可以使用以下可选的参数，如表 3.2 所示。

表 3.2 flags 的其他选项

选项	说明
O_CREATE	当文件不存在时，将建立该文件，此时会用到 open 的第三个参数
O_EXCL	如果同时指定了 O_CREATE，且文件已经存在，则会出错，用此选项可以测试一个文件是否存在，如果不存在，则创建此文件
O_NOCTTY	当文件名（可以包含路径，即第一个参数 pathname）指向一个终端设备，它将不再是进程控制的终端，即使该进程没有一个终端设备
O_TRUNC	如果文件存在，则该文件将被截断，即长度截断为 0。注意，文件没有以写方式打开也可以截断
O_APPEND	文件以追加方式打开 i，每次进行写操作时，文件指针都会被放置到文件末尾
O_NONBLOCK/O_NDELAY	当文件以非阻塞方式打开后，对于 open 及随后的对该文件进行的操作，都会及时返回，而无须进程等待。这对于普通文件和目录文件没有作用，但对于管道等进程间通信的操作很有用
O_SYNC	文件以同步 I/O 方式打开，任何写操作都会使得进程被阻塞，直到物理写动作完成为止

表 3.2 中给出的标志可以混合使用，各标志之间用“|”符号连接。其实第二个参数为 int 型参

数，该数的每位都对应一个操作，符号“|”是将它们按位或，即加起来，使得需要操作的位被置为“1”。


注 意

上面介绍的标志中有一些可以在文件打开后利用 fcntl 函数进行修改，请参考后续章节。

3. open 函数的 mode 参数说明

如果仅仅是需要打开一个文件，可以不使用 open 函数的第三个参数，但是如果充分考虑到文件可能不存在，再次打开之前就需要创建，即此时需要使用 mode 参数。

mode 的参数值如表 3.3 所示。

表 3.3　mode 参数说明

符号	值	含义
S_IRWXU	00700	文件属主有读、写、执行的权限
S_IRUSR(S_IREAD)	00400	文件属主有读权限
S_IWUSR(S_IWRITE)	00200	文件属主有写权限
S_XWUSR(S_IEXEC)	00100	文件属主有执行权限
S_IRWXG	00070	文件组成员有读、写、执行权限
S_IRGRP	00040	文件组成员有读权限
S_IXGRP	00010	文件组成员有执行权限
S_IRWXO	00007	其他用户有读、写、执行的权限
S_IROTH	00004	其他用户有读权限
S_IWOTH	00002	其他用户有写权限
S_IXOTH	00001	其他用户有执行权限

mode 参数支持“或”运算，也就是说可以同时使用上表中的一个或者几个参数，其间可以使用“|”关键字来直接连接或者对其对应的值进行计算之后获得最后的数值进行直接调用。

4. close 函数

close 函数用于关闭一个已经打开的文件，如果关闭成功，则返回 0，否则返回-1。

对 close 函数的标准调用格式说明如下：

```
#include <unistd.h>
int close ( int fd );
```

需要注意的是，当对文件进行打开和关闭操作时，还会对其相关信息产生相应的影响。

- 当打开一个文件时，该文件描述中的引用计数器值加 1，而关闭一个文件时，该文件描述中的引用计数器值减 1。当引用计数器的值减为 0 时，系统调用 close 不仅将释放该文件的描述符，而且也将释放该文件所占的描述表项。

- 关闭一个文件时也释放该进程加在该文件上的所有记录锁。当一个进程终止时，所有的打开文件都由内核自动关闭。很多程序都使用这一功能而不显式地利用 close 关闭打开的文件。
- 当关闭的不是一个普通文件时，可能会产生一些其他的影响。例如，关闭管道文件的一端时，将影响到管道的另一端。

close 的参数为文件描述符，通常来说这个符号为其他函数的返回值，例如 open 函数等。

【例 3.3】打开和关闭文件应用实例

例 3.3 是一个在 Linux 中打开和关闭文件的应用实例，应用代码调用 open 函数在当前的工作目录下以读写打开方式来打开一个名为“opentest”的文件，如果该文件不存在，则创建该文件，创建该文件的时候使用 S_IRWXU 关键字给予该文件读写操作的权限。open 函数将 opentest 的文件描述符返回给一个 int 类型的变量 temp，然后使用 printf 函数将该描述符输出，并且使用 close 函数来关闭文件，其流程如图 3.3 所示。



图 3.3　在 Linux 中打开并且关闭一个文件

实例的应用代码如下：

```
1	//这是一个标准的 open 函数调用实例，打开文件 opentest，如果没有则创建
2	//然后返回文件的描述符，并且关闭文件后退出
3	#include <stdlib.h>
4	#include <fcntl.h>
5	#include <stdio.h>
6	int main(void)
7	{
8	   int fd;                                          //文件描述符
```

```
9         int temp;                                              //临时变量
10        fd = open("./opentest",O_RDWR|O_CREATE,S_IRWXU);       //创建文件 opentest
11        printf("The File Descriptor is %d\n",fd);              //输出文件描述符
12        temp = close(fd);                                      //关闭文件
13        exit(0);                                               //退出
14    }
```

在终端中使用 gcc 对其进行编译并且运行，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter3$ gcc exam301Open.c -o exam301OpenFun
//编译链接，生成可执行文件 exam301OpenFun
alloy@ubuntu:~/linuxc/chapter3$ ./exam301OpenFun      //运行可执行文件 exam301OpenFun
The File Descriptor is 3                              //输出文件描述符 3
```

【例 3.4】打开和关闭指定文件的应用实例

例 3.4 是一个使用 open 函数在当前工作目录下打开或者创建一个文件的应用实例，实例中的文件名是固定的一个字符串，如果用户希望在调用应用代码的时候手动指定（输入）文件名，此时可以利用 main 函数的 argv 参数来传递用户的输入，例 3.4 是一个允许用户输入任意字符串作为打开或者创建文件名称的实例，其流程如图 3.4 所示。

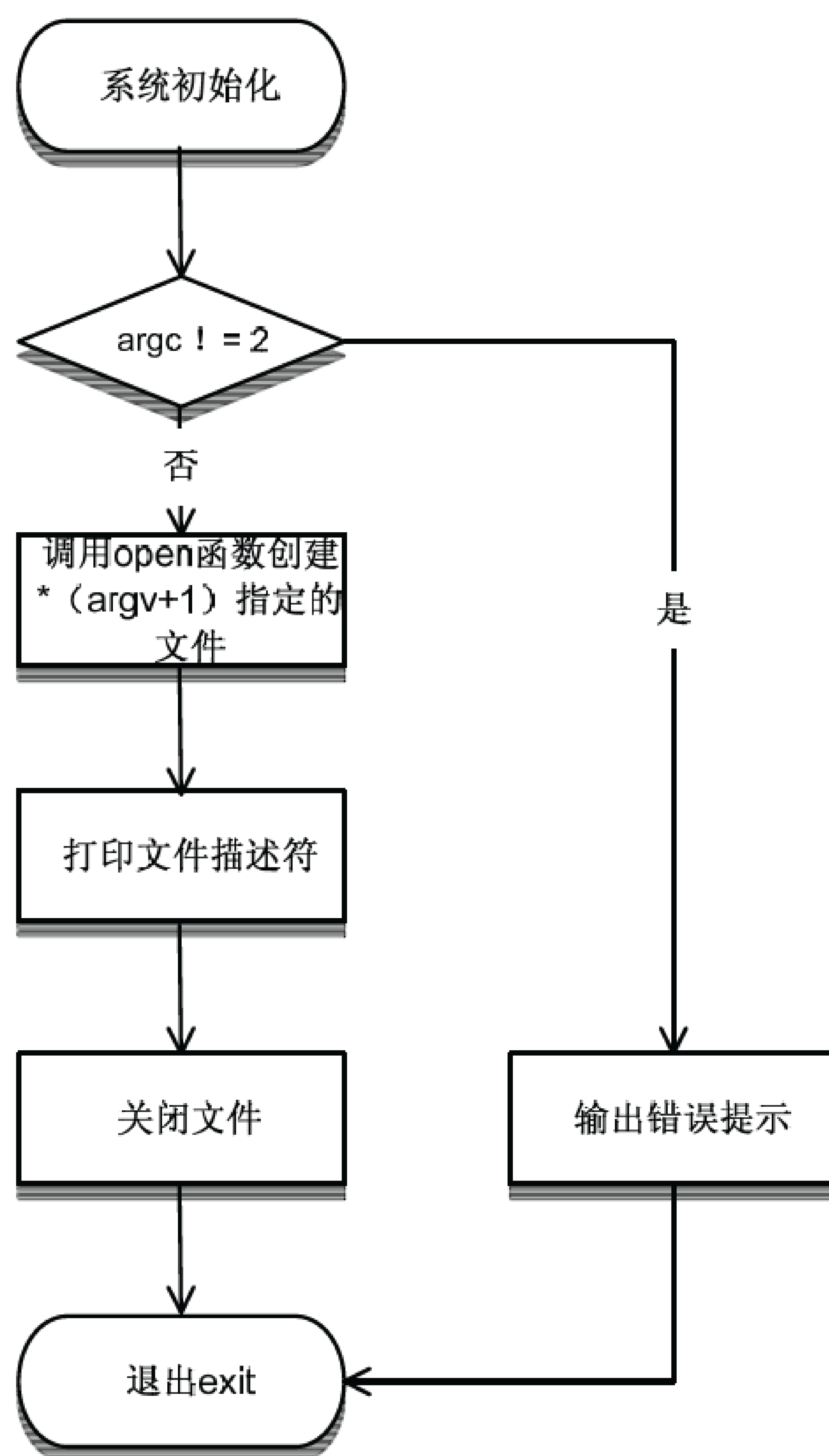


图 3.4 在 Linux 中打开/创建一个任意名称的文件

实例的应用代码如下：

```
1     //这是使用 argv 来传递待打开或者创建文件参数的应用实例
2     //打开或者创建文件成功之后打印文件描述符，并且关闭文件退出
3     //调用文件的格式为“./exam32OpenFun 文件名”
4     #include <stdlib.h>
5     #include <fcntl.h>
6     #include <stdio.h>
7     int main(int argc,char *argv[])              //main 函数的两个标准参数
8     {
9        int fd;                                   //文件描述符
10       int temp;                                 //临时变量
11       if(argc != 2)                             //如果参数不是两个，则说明用户输入参数错误
12       {
13          printf("Plz input the correct file name as './exam302OpenFun filename'\n");
14          //输出提示格式
15       }
16       else      //如果格式正确
17       {
18          fd = open(*(argv+1),O_RDWR|O_CREATE,S_IRWXU);
         //创建 argv 的第 2 个字符开始指定的文件
19          printf("The File Descriptor is %d\n",fd);            //输出文件描述符
20          temp = close(fd);                                    //关闭文件
21       }
22       exit(0);                                                //退出
23    }
```

在终端中使用 gcc 对其编译并且运行，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter3$ gcc exam302open.c -o exam302openFun
//编译链接生成可执行文件 exam302openFun
alloy@ubuntu:~/linuxc/chapter3$ ./exam302openFun
Plz input the correct file name as './exam302OpenFun filename'
//运行 exam302openFun 提示用户输入 filename
alloy@ubuntu:~/linuxc/chapter3$ ./exam302openFun stropentest
The File Descriptor is 3
//在当前目录路径下创建或者打开文件 stropentest，并且返回文件描述符 3
alloy@ubuntu:~/linuxc/chapter3$ ./exam302openFun /home/alloy/linuxc/stropentest
The File Descriptor is 3
在路径/home/alloy/linuxc/下创建或者打开文件 stropentest，并且返回文件描述符 3
```


注　意

从图 3.3 和图 3.4 中可以看到，如果希望在执行文件时手动输入一个字符串作为 open 函数的参数来创建或者打开文件，需要利用 main 函数的 argv 参数，此时应该在传递 argv 参数之前利用 argc 参数来检查 argv 参数的数目是否正确，然后将 argv 参数传递给 open 函数作为文件名。另外需要注意的是 argv 参数可以是一个带路径的字符串。

3.2.2　创建文件

在上一小节中介绍了利用 open 函数创建一个并不存在的文件，但是如果仅仅是想创建一个文件而并不想打开它，此时可以使用 create 函数。

create 函数用于在 Linux 中创建一个文件，如果创建成功，则返回该文件对应的文件描述符，如果出错则返回-1。

对 create 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int create (const char *pathname, mode_t mode);
```

对 create 函数的各个参数和应用实例说明如下。

1. create 函数的 pathname 参数说明

create 函数的 pathname 参数使用方法和 open 函数的 pathname 参数应用方法完全相同，可以参考第 3.2.1 小节。

2. create 函数的 mode 参数说明

create 函数的 mode 参数使用方法和 open 函数中的 mode 参数完全相同，可以参考第 3.2.1 小节。



注 意

create 函数其实等同于 int open(const char *pathname, O_WRONLY|O_CREATE | O_TRUNC, mode_t mode)。

【例 3.5】创建文件应用实例

例 3.5 是 create 函数的应用实例，应用代码使用 main 函数的参数集合的第 2 个参数作为即将创建文件的 pathname 参数，然后在当前目录下建立一个属性为 S_IRWXU 的文件，其流程如图 3.5 所示。

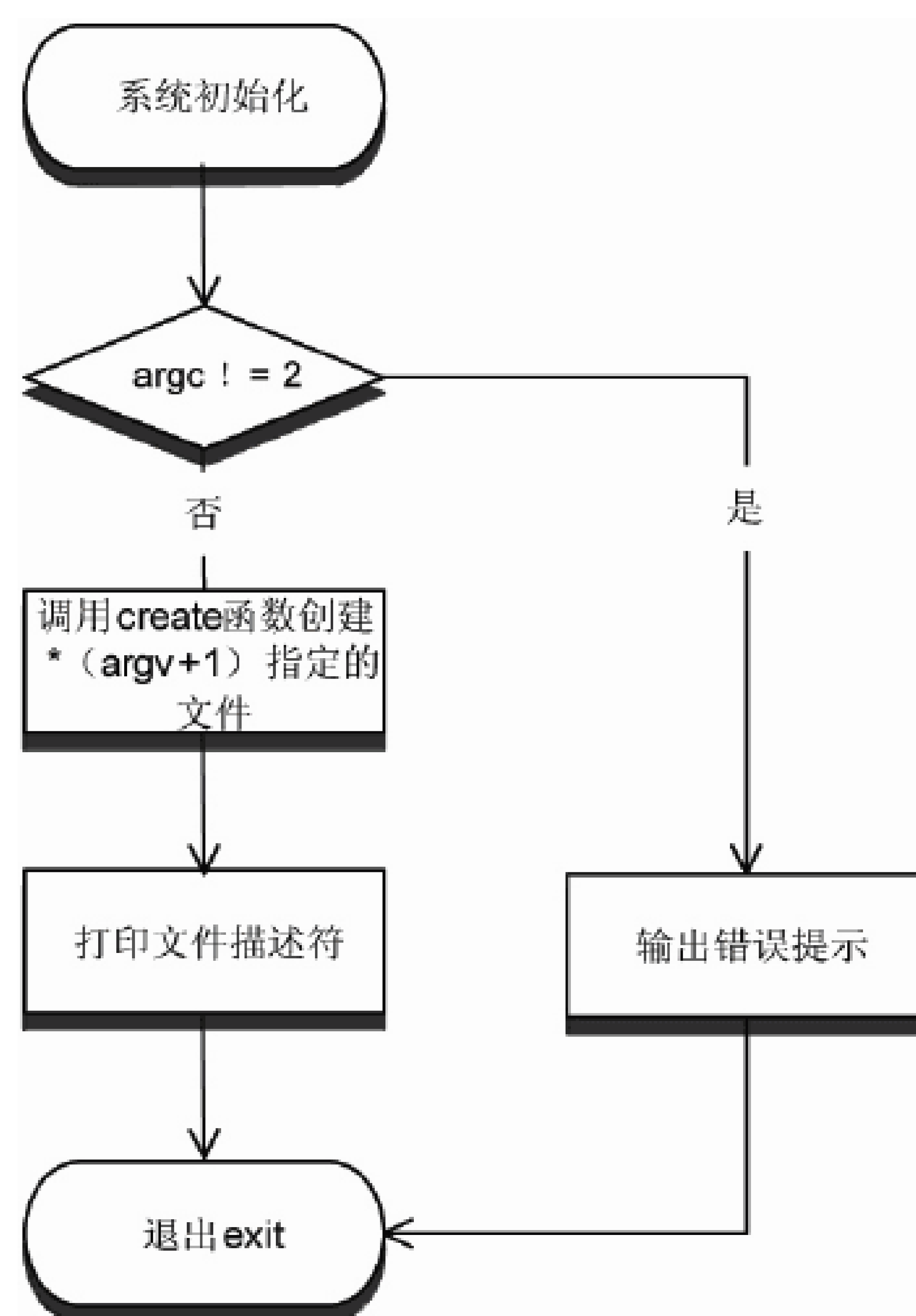


图 3.5　在 Linux 中创建文件

实例的应用代码如下：

```
1       //使用 create 函数来创建文件的应用实例
2       //待创建文件的名称由 argv 参数给出，然后打印文件描述符并且退出
```

```
3   //调用文件的方式为"./exam303CreateFun 文件名”
4   #include <fcntl.h>
5   #include <stdio.h>
6   int main(int argc,char *argv[])
7   {
8     int fd;        //文件描述符
9     if (argc!=2)
10    {
11      printf("Plz input the correct file name as './exam303CreateFun filename'\n");
        //参数错误
12      return 1;                    //退出
13    }
14    else
15    {
16      fd = create(*(argv+1),S_IRWXU); //参数字符串的第 2 个数据作为文件名
17      printf("The File Descriptor is %d\n",fd);
18      return 0;
19    }  //可以通过主函数的返回值来判断执行的状态
20  }
```

在终端中使用 gcc 进行编译并且运行，可以看到如下输出：

```
alloy@ubuntu:~/linuxc/chapter3$ gcc exam303create.c -o exam303createFun
//编译链接生成可执行文件
alloy@ubuntu:~/linuxc/chapter3$ ./exam303createFun
Plz input the correct file name as './exam303CreateFun filename'
//提示参数错误
alloy@ubuntu:~/linuxc/chapter3$ ./exam303createFun createtest
The File Descriptor is 3
//创建文件，并且返回文件描述符
```

3.2.3 将数据写入文件

将数据写入文件是一种最常见的文件应用操作，此时可以调用 write 函数，该函数用于向一个已经打开的文件中写入数据，如果操作成功则返回已经写入的数据字节数，如果操作失败则返回“-1”。

对 write 函数的标准调用格式说明如下：

```
#include <unistd.h>
ssize_t write (int fd, void *buf, size_t count );
```

write 的函数返回值通常与参数 count 的值相同，否则表示出错。write 出错的最常见原因是磁盘已满，或者超过了文件长度限制。



注 意

对于普通文件而言，写操作从文件的当前位移量处开始。如果在打开该文件时，指定了 O_APPEND 选择项，则在每次写操作之前，将文件位移量设置在文件的当前结尾处。在一次成功写之后，该文件位移量增加实际写的字节数。关于 O_APPEND 选择项的相关说明可以参考第 3.2.4 小节。

对 write 函数的各个参数和应用实例说明如下。

1. write 函数的 fd 参数说明

fd 参数是待写入文件的文件描述符，其通常通过 open、create 等函数获得。

2. write 函数的 buf 参数说明

buf 是一个指向写入缓冲区的指针，待写入数据必须存放在该缓冲区内。

3. write 函数的 count 参数说明

count 表示本次操作将要写入文件的数据字节数。

【例 3.6】将数据写入文件应用实例

例 3.6 是将数据写入文件的应用实例，应用代码首先打开参数字符串指定的文件，如果没有则创建这个文件，然后对该文件写入一个字符串“this is a test！”，该字符串存放在缓冲区 writebuf 中，其流程如图 3.6 所示。

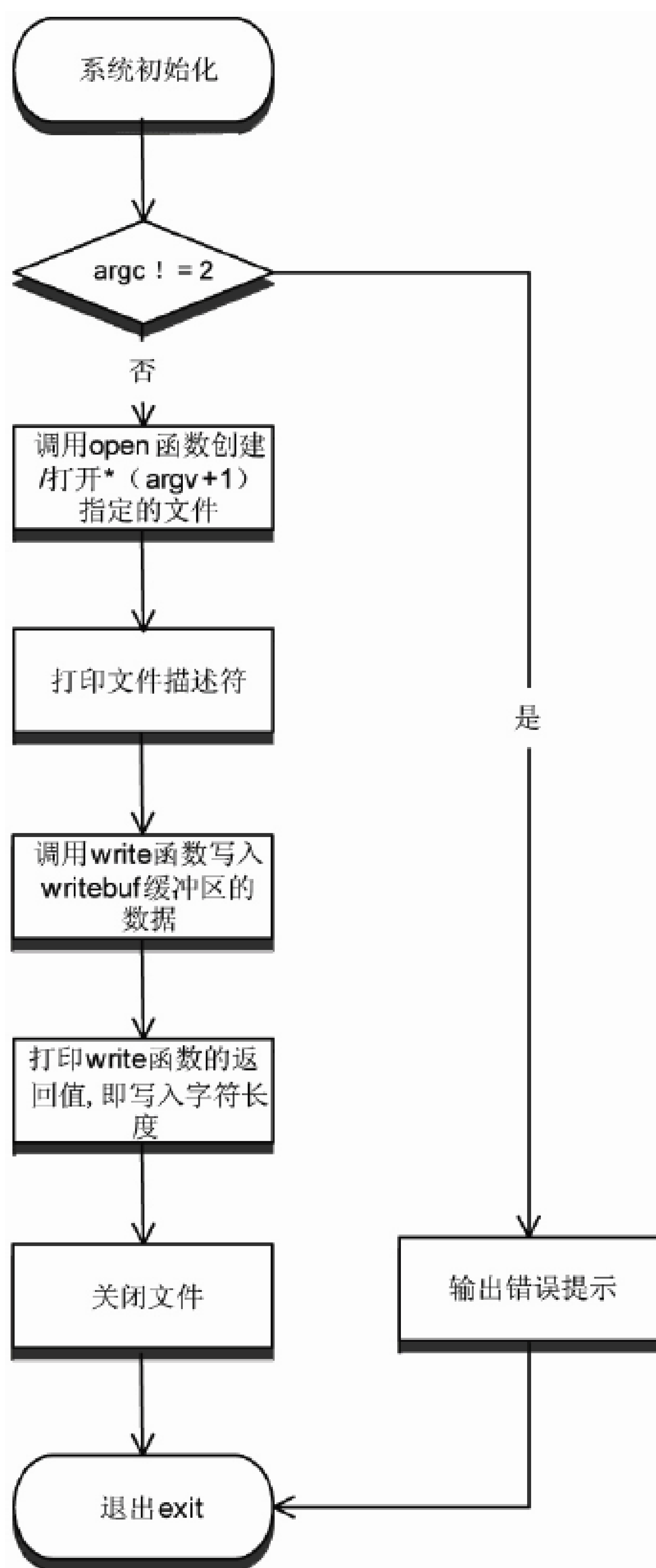


图 3.6　在 Linux 中将数据写入文件

实例的应用代码如下：

```
1   //这是一个标准的将字符串写入文件的应用，待创建的文件名由 argv[1]给出
2   //实例输出创建文件的文件描述符 fd 和写入文件的字符串长度
3   #include <fcntl.h>
4   #include <stdio.h>
5   int main(int argc,char *argv[])
6   {
7       int fd;                                   //文件描述符
8       int temp;                                 //临时变量
9       char writebuf[] = "this is a test!\n";    //存放的待写入数据
10      if(argc != 2)                             //如果参考字符串
11      {
12          printf("Plz input the correct file name as 'exam304WriteFun filename'\n");
13          //输出提示字符串
14          return 1;
15      }
16      else
17      {
18          fd = open(*(argv+1),O_RDWR|O_CREATE,S_IRWXU);
19        //打开文件，如果没有则创建
20      }
21      printf("The File Descriptor is %d\n",fd);  //打印文件描述符
22      temp = write(fd,writebuf,17);              //使用文件描述符调用文件
23      printf("The input length is %d\n",temp);
24      close(fd);
25      return 0;
26  }
```

在终端中使用 gcc 对其进行编译并且运行，可以看到如下输出：

```
alloy@ubuntu:~/linuxc/chapter3$ gcc exam304write.c -o exam304writeFun
//编译链接生成可执行文件
alloy@ubuntu:~/linuxc/chapter3$ ./exam304writeFun writetest
The File Descriptor is 3
The input length is 17
//将字符串写入文件 writetest，并且返回文件描述符和写入数据长度
alloy@ubuntu:~/linuxc/chapter3$ cat -n writetest
1    this is a test!
2    alloy@ubuntu:~/linuxc/chapter3$
//使用 cat –n 命令查看文件 writetest 中的数据
```

【例 3.7】将数据写入文件进阶操作应用实例

在例 3.6 中调用 write 函数将 writebuf 中的字符串写入文件的时候，write 函数表明当次写入字符数目的参数 count 是预先计算好，直接将对应的数值“17”传递给函数的，但是在实际应用中，字符串的长度不能预先计算，此时可以使用 strlen 函数或者 sizeof 运算符来获得待写入字符串的长度。

sizeof 是运算符，其返回值在编译时得到，参数可以是数组、指针、类型、对象、函数等，其功能是获得保证能容纳实现所建立的最大对象的字节大小。由于在编译时计算，因此 sizeof 不能用来返回动态分配的内存空间大小。实际上，用 sizeof 来返回类型以及静态分配的对象、结构或数组

所占的空间，返回值与对象、结构、数组所存储的内容没有关系；如下参数给出了 sizeof 返回的值表示的含义。

- 数组：对应编译时分配的数组空间大小。
- 指针：存储该指针所用的空间大小。
- 类型：该类型所占的空间大小。
- 对象：对象的实际占用空间大小。
- 函数：函数的返回类型所占的空间大小，需要注意的是函数返回值不能是 void。

strlen 是函数，用于返回参数字符串的长度，在被调用时返回空间大小，其参数必须是字符型指针（char*），当使用数组名作为参数传入时，数组被转换为指针，strlen 函数的具体使用方法可以参考第 2.7.2 小节。

strlen 参数所对应的字符串可能是用户定义的，也可能是内存中随机的，其实际完成的功能是从代表该字符串的第一个地址开始遍历，直到遇到结束符 NULL，返回的字符串长度大小不包括 NULL。

例 3.7 是一个使用 strlen 函数来获得当前写入字符串长度传递给 write 函数作为 count 参数的实例，其流程如图 3.7 所示。

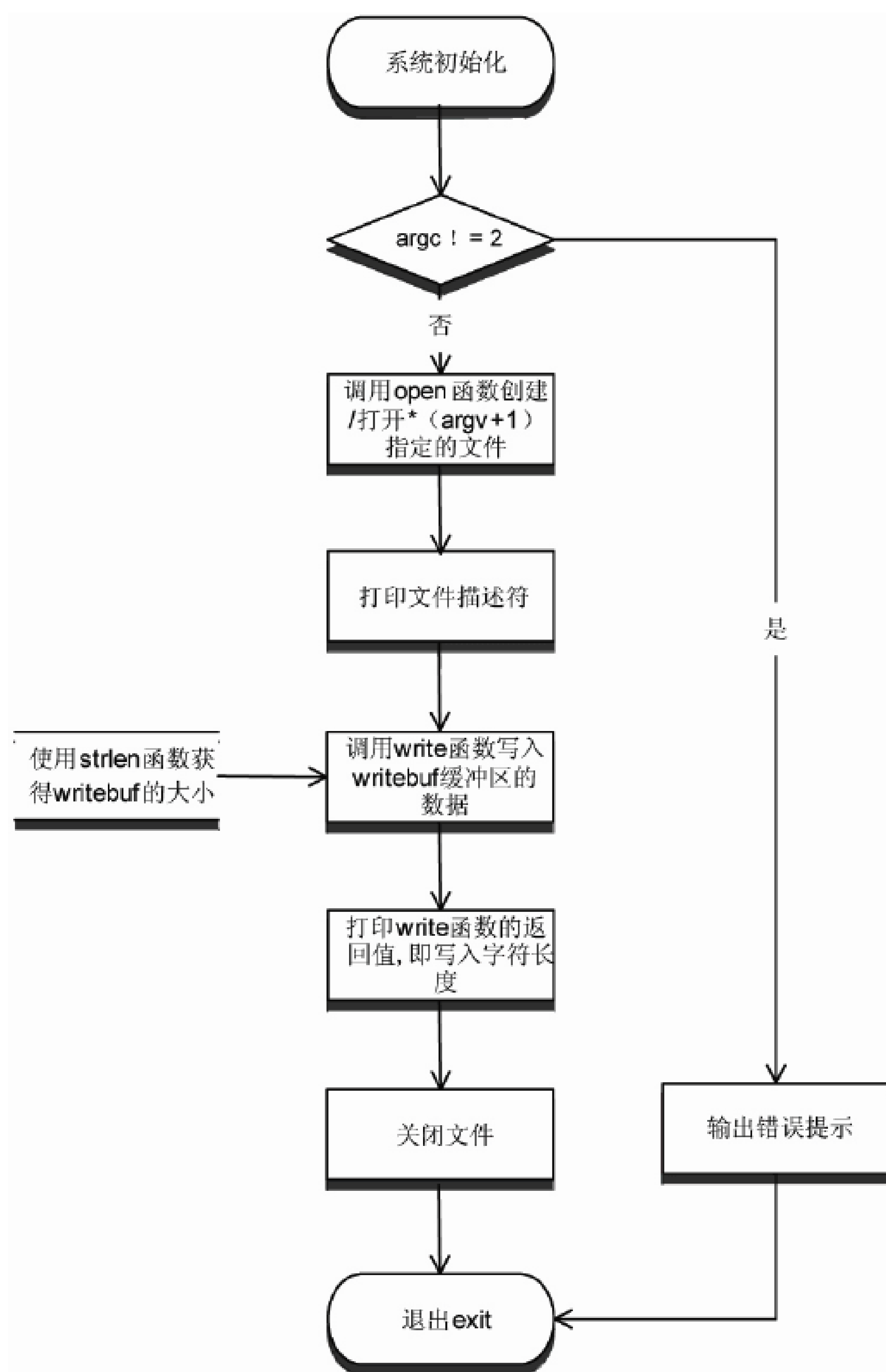


图 3.7　使用 strlen 获得 write 的 count 参数

实例的应用代码如下：

```
1	//这是一个标准的将字符串写入文件的应用，待创建的文件名由 argv[1]给出
2	//写入字符串的长度不是一个固定值，而是由 strlen 函数返回
3	//实例输出创建文件的文件描述符 fd 和写入文件的字符串长度
4	#include <fcntl.h>
5	#include <stdio.h>
6	#include <string.h>
7	int main(int argc,char *argv[])
8	{
9	   int fd;                                    //文件描述符
10	   int temp;                                  //临时变量
11	   char writebuf[] = "this is a test!\n";     /存放的待写入数据
12	   if(argc != 2)                              //如果参考字符串
13	   {
14	      printf("Plz input the correct file name as 'exam305WriteFun filename'\n");
15	      //输出提示字符串
16	      return 1;
17	   }
18	   else
19	   {
20	      fd = open(*(argv+1),O_RDWR|O_CREATE,S_IRWXU);
21	    //打开文件，如果没有则创建
22	   }
23	   printf("The File Descriptor is %d\n",fd);        //打印文件描述符
24	   temp = write(fd,writebuf,strlen(writebuf));      //使用文件描述符调用文件
25	   printf("The input length is %d\n",temp);
26	   close(fd);
27	   return 0;
28	}
```

在终端中使用 gcc 对其进行编译并且运行，可以看到和例 3.6 类似的运行结果，在此不再赘述。

【例 3.8】将用户输入的字符串写入文件的应用实例

例 3.6 和例 3.7 在将字符串写入文件的时候都是将字符串预先存放到写入缓冲区 writebuf 中，那么如果想将用户当前输入的字符串写入文件该怎么办呢？第一种方法是参考例 3.4 创建一个用户输入名称的文件，使用 main 函数的 argv 参数来传递待写入的字符串，由于 argv 的第 1 个参数*(argv+1)已经用于传递文件名称，所以此时可以使用第 2 个参数*(argv+2)用于传递待写入字符串，这个字符串必须以空格结尾，其中不能有空格。

例 3.8 是一个将用户输入的字符串写入用户指定文件的实例，其流程如图 3.8 所示，在代码中利用了 strlen 函数来获得待写入的字符串长度。

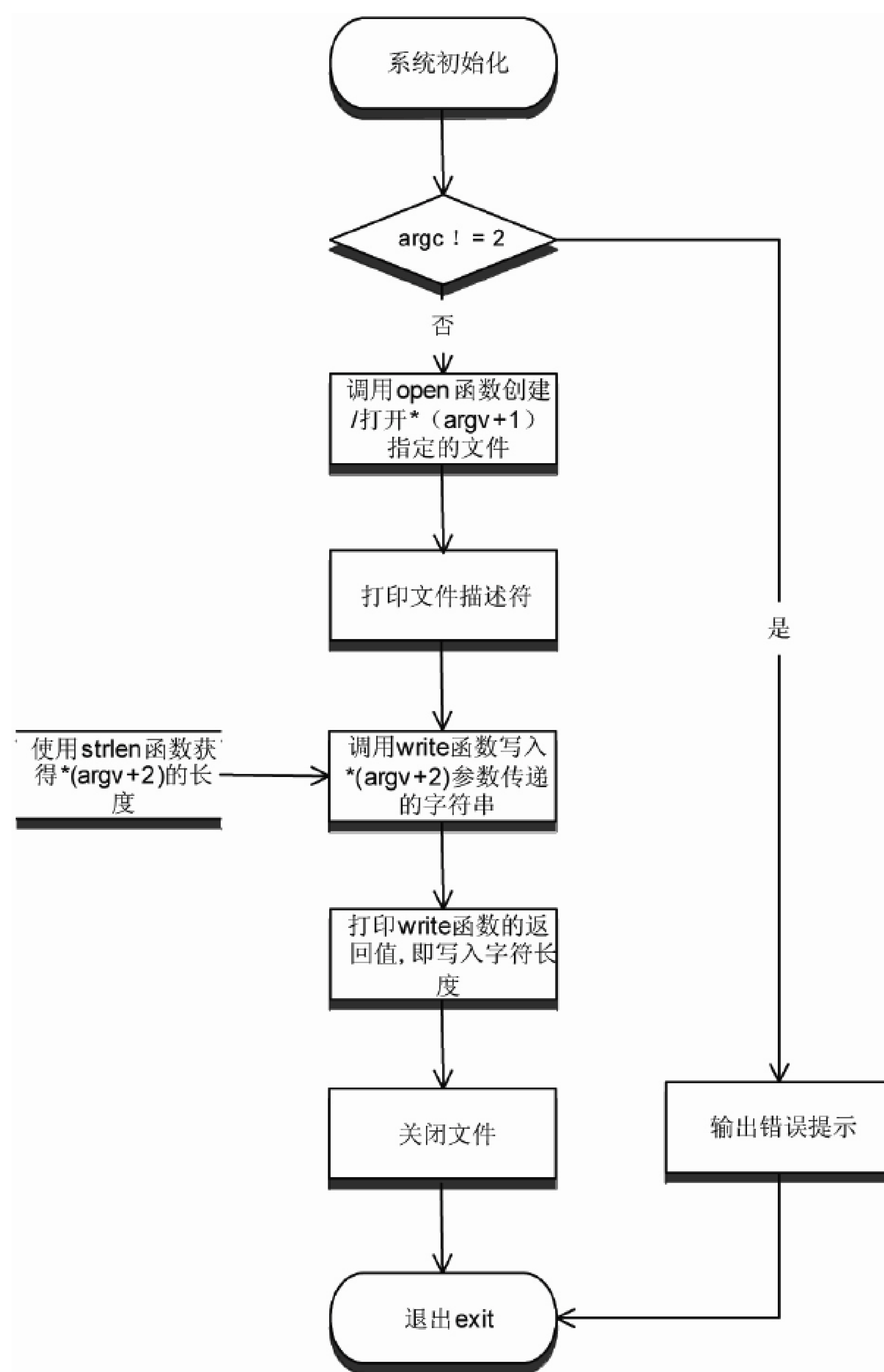


图 3.8　将用户输入的字符串写入文件

实例的应用代码如下：

```
1     //应用实例使用了 argv[2]传送待写入的数据，但是不能有空格
2     //输入格式为 exam306WriteFun + 文件名称 + 待写入字符串
3     #include <fcntl.h>
4     #include <stdio.h>
5     #include <string.h>
6     int main(int argc,char *argv[])
7     {
8        int fd;                          //文件描述符
9        int temp;                        //临时变量
10       if(argc != 3)                    //如果参考字符串错误
11       {
12          printf("Plz input the correct file name as 'exam306WriteFun filename string'\n");
13          //输出提示字符串
14          return 1;
15       }
```

```
16          else
17          {
18              fd = open(*(argv + 1),O_RDWR|O_CREATE,S_IRWXU);
19            //打开文件，如果没有则创建
20          }
21          printf("The File Descriptor is %d\n",fd);               //打印文件描述符
22          temp = write(fd,*(argv + 2),strlen(*(argv+2)));         //使用文件描述符调用文件
23          printf("The input length is %d\n",temp);
24          close(fd);
25          return 0;
26      }
```

在终端中使用 gcc 编译并且运行，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter3$ gcc exam306write.c -o exam306writeFun
//编译链接生成可执行文件
alloy@ubuntu:~/linuxc/chapter3$ ./exam306writeFun strwritetest myfirstinput
The File Descriptor is 3
The input length is 12
//其中 strwritetest 对应*(argv+1)，myfirstinput 对应*(argv+2)，后者中不能有
//空格，输出文件描述符和输入的字符串长度
alloy@ubuntu:~/linuxc/chapter3$ cat -n strwritetest
 1      myfirstinputalloy@ubuntu:~/linuxc/chapter3$
//使用 cat-n 命令来查看 strwritetest 文件内容
```


注 意

在实际应用中基本上没有人使用这种方法来传递待写入的数据，因为有很多缺陷，例如*(argv+2)传递的字符串中不能有空格等，但是如果待写入的数据很短小，或者干脆就是一些辅助参数，还是可以使用这种方法的。

【例 3.9】将用户输入的字符串写入文件的进阶应用实例

在实际应用中通常不会有人利用 main 函数的参数来传递待写入的字符串，通常是使用 gets 函数来获得用户从键盘输入的字符串，这个字符串必须以回车换行结尾并且其中不能有回车换行，可以有空格，关于 gets 函数的详细使用方法将会在第 6 章进行详细介绍。

例 3.9 是一个使用 gets 函数来获得用户输入字符串的实例，其流程如图 3.9 所示。

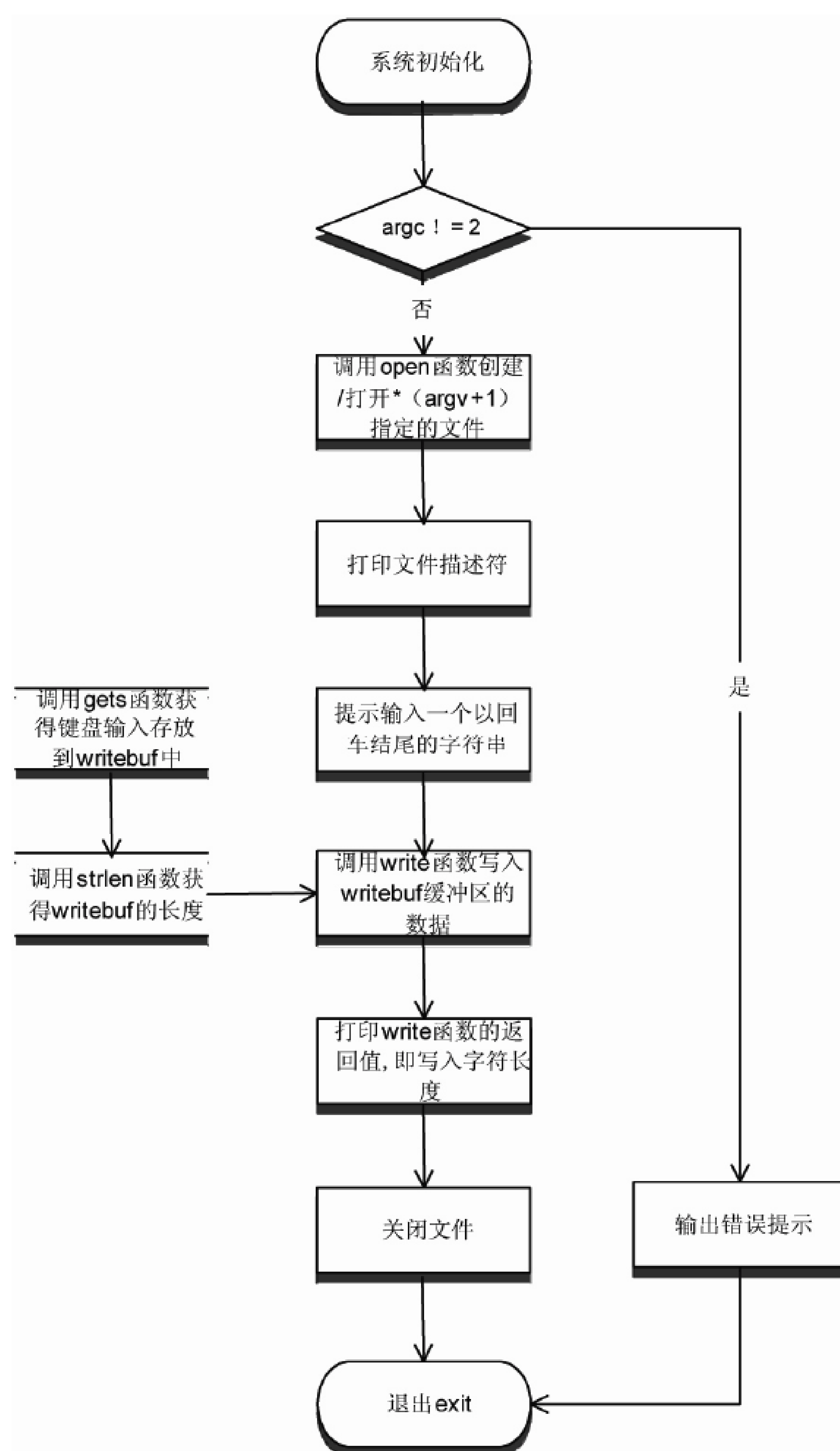


图 3.9　将 gets 函数返回的用户键盘输入写入文件

实例的应用代码如下：

```
1    //使用 gets 函数从标准输入（键盘）获得一个以回车换行为结束的字符串，可以带空格
2    //运行时屏幕会提示输入字符串，以回车键结尾
3    //需要注意的是待输入的字符串存放在 writebuf 中，不能超过 30 个字节并且不会带回车键
4    #include <fcntl.h>
5    #include <stdio.h>
6    #include <string.h>
7    int main(int argc,char *argv[])
8    {
9      int fd;                          //文件描述符
10     int temp;                        //临时变量
11     char writebuf[30];               //用于存放写入字符串
12     if(argc != 2)                    //如果参考字符串错误
13     {
14       printf("Plz input the correct file name as 'exam307WriteFun filename'\n");
```

```
15            //输出提示字符串
16            return 1;
17          }
18          else
19          {
20            fd = open(*(argv + 1),O_RDWR|O_CREATE,S_IRWXU);
21          //打开文件，如果没有则创建
22          }
23          printf("The File Descriptor is %d\n",fd);              //打印文件描述符
24          printf("Plz input the strings and use Enter as the end!\n");
25          gets(writebuf);                                         //将终端输入的数据写入文件
26          temp = write(fd,writebuf,strlen(writebuf));             //使用文件描述符调用文件
27          printf("The input length is %d\n",temp);
28          close(fd);
29          return 0;
30      }
```

在终端中使用 gcc 编译并且运行，可以看到如下输出：

```
alloy@ubuntu:~/linuxc/chapter3$ gcc exam307write.c -o exam307writeFun
//编译链接生成可执行文件
alloy@ubuntu:~/linuxc/chapter3$ ./exam307writeFun strgetwritetest
The File Descriptor is 3
Plz input the strings and use Enter as the end!
//运行，目标文件为 strgetwritetest，提示输入一个字符串
this is a test!
//用户输入的字符串
The input length is 15
//返回写入的字符长度
alloy@ubuntu:~/linuxc/chapter3$ cat -n strgetwritetest
this is a test!alloy@ubuntu:~/linuxc/chapter3$
//使用 cat –n 命令来查看 strgetwritetest 文件的内容
```

【例 3.10】带回车换行的写入字符串应用实例

从例 3.9 的运行输出中可以发现一个问题——写入文件的字符串并没有利用回车键换行，这是因为 gets 函数是以回车换行来判断用户的字符串输入是否结束的，若要解决这个问题，可以在 writebuf 之后加上回车换行符即可。例 3.10 即为一个带回车换行的字符串写入实例，流程如图 3.10 所示。

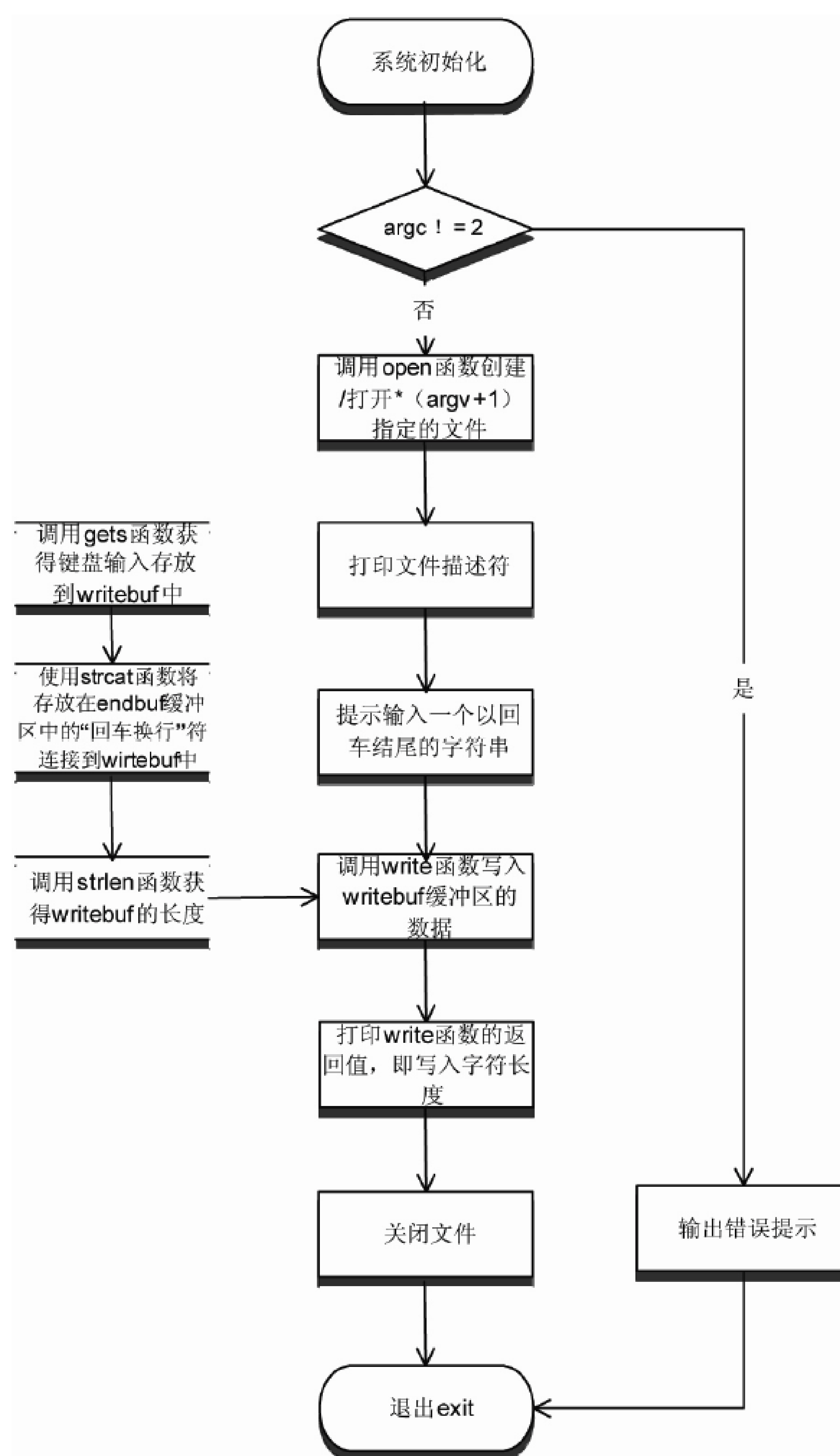


图 3.10　写入带回车换行的字符串

在实例中定义了一个 enterbuf 缓冲区用于存放回车换行符“\n”，然后使用 strcat 函数将这个缓冲区和 writebuf 缓冲区连接，此时就将一个回车换行符添加到了 gets 函数获得字符串之后，然后将其写入文件。

实例的应用代码如下：

```
1     //这是在上一个实例的基础上用 strcat 函数解决了回车换行的问题
2     #include <fcntl.h>
3     #include <stdio.h>
4     #include <string.h>
5     int main(int argc,char *argv[])
6     {
7        int fd;                          //文件描述符
8        int temp;                        //临时变量
9        char writebuf[30];               //用于存放写入字符串
10       char endbuf[] = "\n";            //存放一个回车换行符
```

```
11          if(argc != 2)                          //如果参考字符串错误
12          {
13             printf("Plz input the correct file name as 'exam308WriteFun filename'\n");
14             //输出提示字符串
15             return 1;
16          }
17          else
18          {
19             fd = open(*(argv + 1),O_RDWR|O_CREATE,S_IRWXU);
20           //打开文件，如果没有则创建
21          }
22          printf("The File Descriptor is %d\n",fd);          //打印文件描述符
23          printf("Plz input the strings!\n");
24          gets(writebuf);                                    //将终端输入的数据写入文件
25          strcat(writebuf,endbuf);                           //添加换行符
26          temp = write(fd,writebuf,strlen(writebuf));        //使用文件描述符调用文件
27          printf("The input length is %d\n",temp);
28          close(fd);
29          return 0;
30       }
```

在终端中使用 gcc 对其进行编译并且运行，可以看到和例 3.9 类似的运行结果，再次使用“cat –n”来查看写入字符串的文件内容，可以看到已经写入了回车换行。

3.2.4 在文件中进行定位操作

在第 3.2.3 小节中介绍使用 write 函数将数据写入文件的时候都是一次性地将需要写入的数据写入文件中，如果需要分多次将数据写入，则需要关心文件的偏移量，简而言之就是需要知道上一次的数据都写到了文件的什么位置，下一次的数据要从什么地方开始接着写入。

在 Linux 中，每个打开的文件都有一个与其相关联的当前文件偏移量（也叫文件指针），它通常是一个非负整数，用以度量从文件开始处计算的字节数。通常情况下，读、写操作都从当前文件偏移量处开始，并使偏移量增加所读写的字节数。

lseek 函数用于指定文件偏移量的位置，从而实现文件的随机存取，如果操作成功则返回新的文件偏移量，如果出错则返回-1。

对 lseek 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fds, off_t offset, int whence);
```

注意：off_t 类型在 Linux 中通常就是 long 类型，其默认为一个 32 位的整数，在 gcc 编译中会被编译为 long int 类型，在 64 位的 Linux 系统中则会被编译为 long long int，这是一个 64 位的整数，其定义在 unistd.h 头文件中，定义为如下形式：

```
1     # ifndef __off_t_defined
2     #   ifndef __USE_FILE_OFFSET64
3     typedef __off_t off_t;
```

```
4	#	else
5	typedef __off64_t off_t;
6	#	endif
7	#	define __off_t_defined
8	# endif
9	# if defined __USE_LARGEFILE64 && !defined __off64_t_defined
10	typedef __off64_t off64_t;
11	#	define __off64_t_defined
12	# endif
13	/* Move FD's file position to OFFSET bytes from the
14		beginning of the file (if WHENCE is SEEK_SET),
15		the current position (if WHENCE is SEEK_CUR),
16		or the end of the file (if WHENCE is SEEK_END).
17		Return the new file position. */
18	#ifndef __USE_FILE_OFFSET64
19	extern __off_t lseek (int __fd, __off_t __offset, int __whence) __THROW;
20	#else
21	# ifdef __REDIRECT_NTH
22	extern __off64_t __REDIRECT_NTH (lseek,
23						(int __fd, __off64_t __offset, int __whence),
24						lseek64);
25	# else
26	#	define lseek lseek64
27	# endif
28	#endif
29	#ifdef __USE_LARGEFILE64
30	extern __off64_t lseek64 (int __fd, __off64_t __offset, int __whence)
31		__THROW;
32	#endif
```

1. lseek 函数的 fds 参数说明

fds 参数是待写入文件的文件描述符，其通常通过 open、create 等函数获得。

2. lseek 函数的 offset 参数说明

offset 是文件偏移量，指的是每一次对文件的读写操作所需移动的距离，单位为字节；offset 的取值可正可负，其正值指的是向前移，负值指的是向后移。


注 意

对于普通文件而言，offset 通常都是正值，所以在使用的时候最好能先测试其值以确保取值。

3. lseek 函数的 whence 参数说明

whence 有三种不同的取值。

- SEEK_SEK：设置偏移量为文件开始位置之后的 offset 个字节。
- SEEK_CUR：设置偏移量为当前偏移量之后的 offset 个字节。

● SEEK_END：设置偏移量为当前文件长度加上的 offset 个字节。

lseek 函数允许文件的偏移量被设置到超过文件结束符（EOF）处，然后在下一次调用 write 时，可以将文件的长度延伸到所需的长度，并用无意义的字符填充这个空隙。如果随后的 read 读取这个空隙间的数据，将得到无意义的值，直到这个文件数据块被真正写回到磁盘上，再读取这个空隙间的数据时将得到 0。这是因为，当在文件尾之后执行 write 函数的话，Linux 系统并不存储无用的数据块。在 read 函数读到该数据块时，系统为 read 函数产生一个全为 0 的数据块，返回给 read 函数。如果一个 read 函数置于文件尾或文件尾之后的文件偏移量，则产生 0 作为 read 的返回值。

另外，由于 lseek 成功执行时返回新的文件位移量，为此可以用下列方式确定一个打开文件的当前位移量：

```
off_t currpos;                          /*定义变量 currpos 的数据类型为 off_t */
currpos = lseek (fd, 0, SEEK_CUR);      /*offset 值为 0*/
```



注　意

可用来确定所涉及的文件是否可以设置位移量，如果文件描述符引用的是一个管道或 FIFO，则 lseek 返回-1，并将 errno 设置为 EPIPE。

【例 3.11】使用偏移量来分次写入数据的应用实例

例 3.11 是一个使用 lseek 函数来分次向文件写入数据的应用实例，应用代码首先打开参数字符串指定的文件，如果没有则打开这个文件，然后对该文件连续写入字符串“this is a test！”且回车换行，该字符串存放在缓冲区 writebuf 中，在每次写入之前都需要将文件偏移量移动到下一次待写入的位置，其流程如图 3.11 所示。

实例的应用代码如下：

```
1    //这是一个使用 lseek 在一个文件中连续写入字符串的应用
2    #include <fcntl.h>
3    #include <stdio.h>
4    #include <string.h>
5    int main(int argc,char *argv[])
6    {
7       int temp,seektemp,i,j;
8       int fd;                                                   //文件描述符
9       char writebuf[17] = "this is a test!\n";                  //字符串最后加上回车换行
10      if(argc!= 2)                                              //如果参数错误
11      {
12          printf("Plz input the corrcet file name as './exam309lseekFun filename string'!\n");
13          return 1;                                             //如果参数不正确则退出
14      }
15      fd = open(*(argv+1),O_RDWR|O_CREATE,S_IRWXU);             //打开文件，如果没有则创建
16      temp = write(fd,writebuf,sizeof(writebuf));               //写入数据
17      seektemp = lseek(fd,0,SEEK_CUR);                          //获得当前的偏移量
18      for(i=0;i<10;i++)                                         //连续写入 10 个字符串
19      {
20         j = sizeof(writebuf) * (i+1);                          //计算下一次的偏移量
```

```
21              seektemp = lseek(fd,j,SEEK_SET);
22              temp = write(fd,writebuf,strlen(writebuf));                //写入数据
23          }
24          close(fd);                                                     //关闭文件
25          return 0;
26      }
```

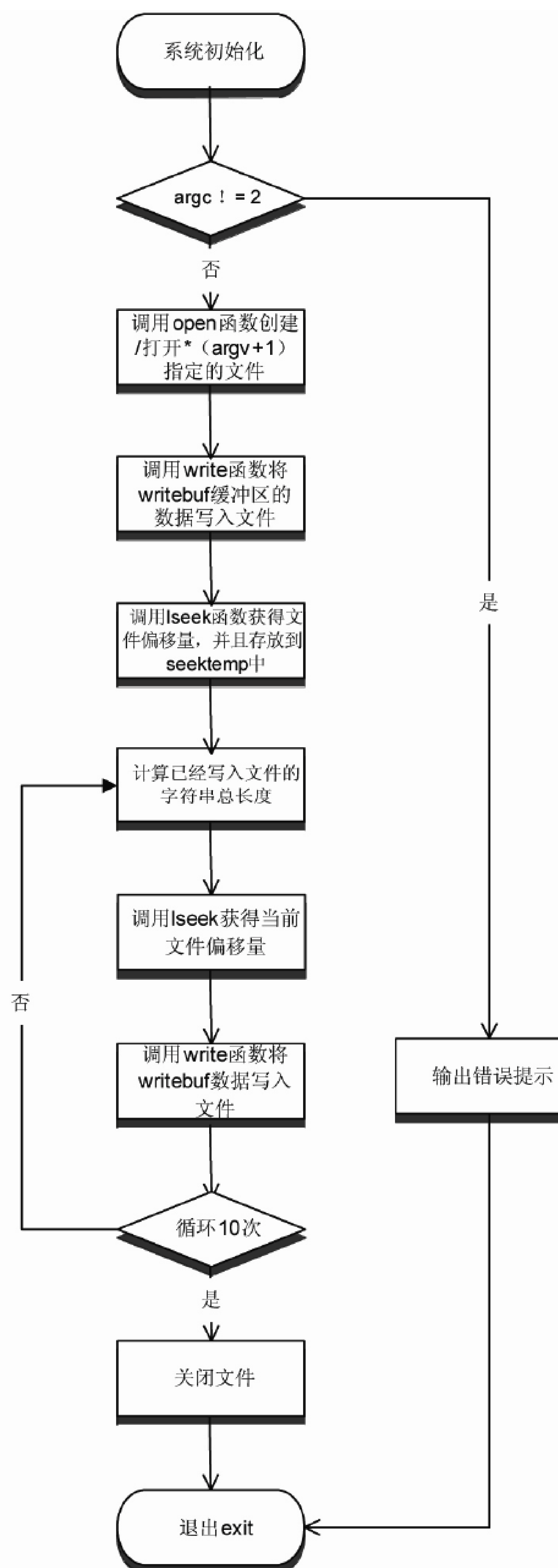


图 3.11　使用偏移量分多次向文件写入数据

在终端中使用 gcc 进行编译并且运行，可以看到如下输出：

```
alloy@ubuntu:~/linuxc/chapter3$ gcc exam309lseek.c -o exam309lseekFun
//编译链接生成可执行文件
alloy@ubuntu:~/linuxc/chapter3$ ./exam309lseekFun strlseektest
//运行，将字符串写入 strlseektest 文件中
alloy@ubuntu:~/linuxc/chapter3$ cat -n strlseektest
 1     this is a test!
 2     this is a test!
 3     this is a test!
 4     this is a test!
 5     this is a test!
 6     this is a test!
 7     this is a test!
 8     this is a test!
 9     this is a test!
10     this is a test!
11     this is a test!
//使用“cat –n”命令查看 strlseektest 文件的内容
```

【例 3.12】使用偏移量来分次写入用户输入的字符串

例 3.10 是一个完整的可以将用户输入字符串写入文件的实例，例 3.11 是一个调用 lseek 函数分次将一个字符串写入文件的实例，将这两者综合起来，就是一个可以将用户的输入连续写入文件的实例，如例 3.12 所示。

例 3.12 首先打开或者创建一个用户指定的文件，文件名由*(argv+1)参数传递，然后调用 gets 函数以获得用户输入，添加回车换行键后写入文件，并且调用 lseek 函数对偏移量进行处理，其流程如图 3.12 所示。

实例的应用代码如下：

```
 1     //这是一个使用 lseek 在一个文件中连续写入用户输入字符串的应用
 2     //用户输入的字符串由 gets 函数获取
 3     //这个地方必须用 strlen 函数而不是 sizeof，前者是缓冲区的实际大小，而后者是缓冲区大小
 4     #include <fcntl.h>
 5     #include <stdio.h>
 6     #include <string.h>
 7     int main(int argc,char *argv[])
 8     {
 9        int temp,seektemp,i,j;
10        int fd;                                   //文件描述符
11        char writebuf[30];                        //用于存放待写入的数据，最长为 30 字节
12        char endbuf[] = "\r\n";                   //用于存放回车换行
13        if(argc!= 2)                              //如果参数错误
14        {
15            printf("Plz input the corrcet file name as './exam310lseekFun filename string'!\n");
16            return 1;                             //如果参数不正确则退出
17        }
18        fd = open(*(argv+1),O_RDWR|O_CREATE,S_IRWXU);  //打开文件，如果没有则创建
19        printf("Plz input the string and use Enter as the end!\n");  //提示输入数据
20        gets(writebuf);                                //获得字符串
21        strcat(writebuf,endbuf);                       //连接回车换行
22        temp = write(fd,writebuf,strlen(writebuf));    //写入数据
```

```
23          seektemp = lseek(fd,0,SEEK_CUR);                         //获得当前的偏移量
24          for(i=0;i<10;i++)                                        //连续写入 10 个字符串
25          {
26              printf("%d,Plz input the string and use Enter as the end!\n",i);  //提示输入数据
27              gets(writebuf);                                      //获得字符串
28              strcat(writebuf,endbuf);                             //连接回车换行
29              j = sizeof(writebuf) * (i+1);                        //计算下一次的偏移量
30              seektemp = lseek(fd,j,SEEK_SET);
31              temp = write(fd,writebuf,strlen(writebuf));          //写入数据
32          }
33          close(fd);                                               //关闭文件
34          return 0;
35      }
```

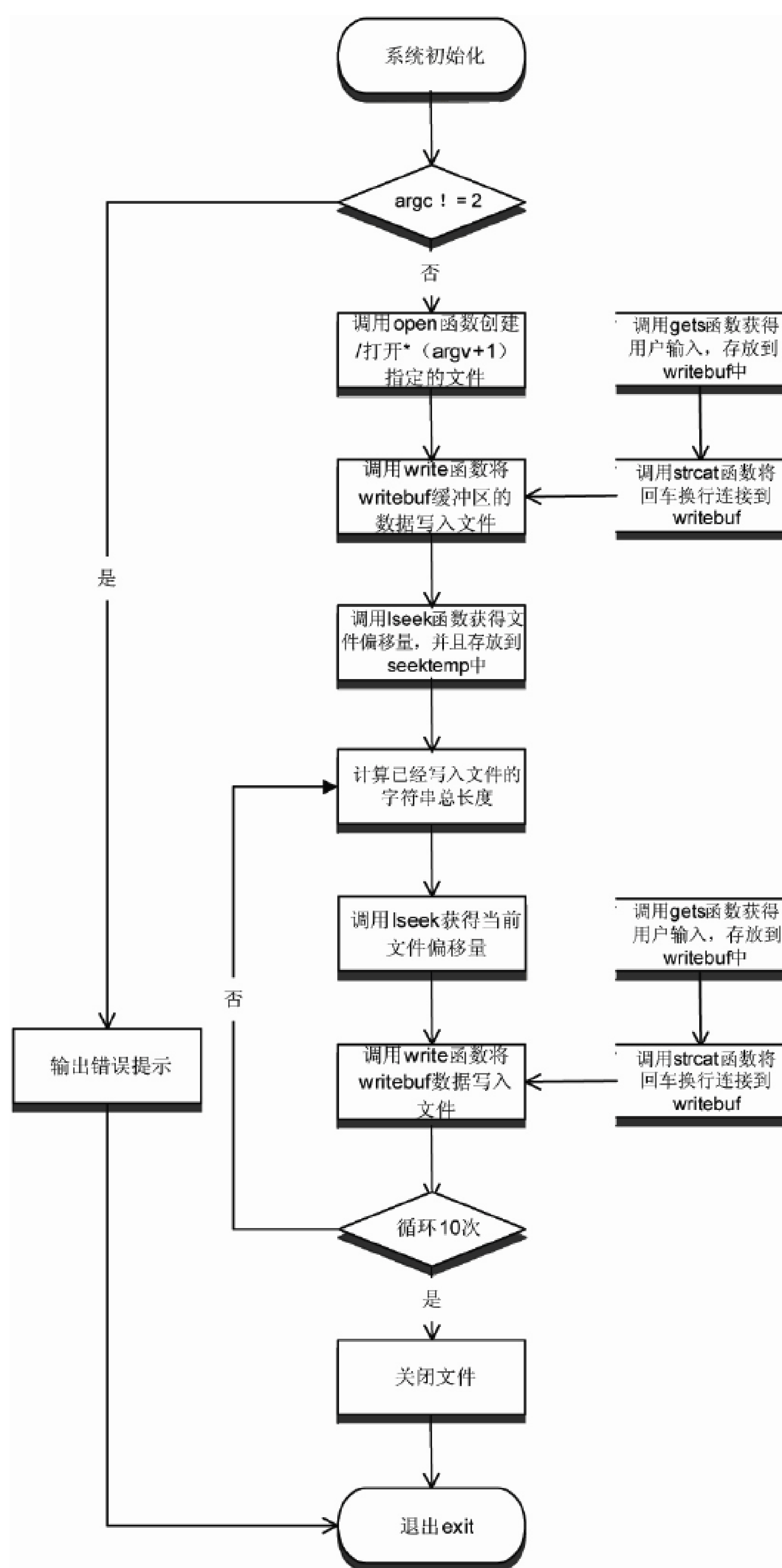


图 3.12　使用偏移量来分次写入用户输入的字符串

在终端中使用 gcc 进行编译并且运行，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter3$ gcc exam310lseek.c -o exam310lseekFun
//编译链接生成可执行文件
alloy@ubuntu:~/linuxc/chapter3$ ./exam310lseekFun lseektest
//打开或者创建 lseektest 文件，然后提示用户输入字符串，这个字符串可以是空格和汉字
Plz input the string and use Enter as the end!
this is a test!
0,Plz input the string and use Enter as the end!
step1
1,Plz input the string and use Enter as the end!
step2
2,Plz input the string and use Enter as the end!
this is step3
3,Plz input the string and use Enter as the end!
step4 = #$#$^
4,Plz input the string and use Enter as the end!

5,Plz input the string and use Enter as the end!
12345678
6,Plz input the string and use Enter as the end!
4.2r645
7,Plz input the string and use Enter as the end!
这是一个测试
8,Plz input the string and use Enter as the end!
this is a test!
9,Plz input the string and use Enter as the end!
this is a test!
//使用 "cat –n" 命令查看文件内容
alloy@ubuntu:~/linuxc/chapter3$ cat -n lseektest
 1     this is a test!
 2     step1
 3     step2
 4     this is step3
 5     step4 = #$#$^
 6
 7     12345678
 8     4.2r645
 9     这是一个测试
10     this is a test!
11     this is a test!
```

3.2.5　从文件中读出数据

既然可以将数据写入到 Linux 的文件中，当然也可以从一个 Linux 文件中读出数据，此时需要调用 read 函数，其从一个已打开的 Linux 文件中读取指定长度的数据，如果操作成功，则返回读到的字节数；如果已经到达了文件的末端则返回 0；如果出错则返回-1。

对 read 函数的标准调用格式说明如下：

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t count) ;
```

通常来说，read 函数的读操作是从文件的当前位移量处开始，在成功返回之前，该位移量增加实际读到的字节数，但是有如下的几种情况可使实际读到的字节数少于要求读的字节数：

- 当读普通文件时，在读到要求字节数之前已到达了文件尾端。例如，若在到达文件尾端之前还有 30 个字节，而要求读 100 个字节，则 read 返回 30，下一次再调用 read 时，它将返回 0（文件尾端）。
- 当从终端设备读时，通常一次最多读一行。
- 当从网络读时，网络中的缓冲机构可能造成返回值小于所要求读的字节数。
- 某些面向记录的设备，例如磁带，一次最多返回一个记录。

对 read 函数的各个参数和应用实例说明如下。


注 意

size_t 类型用于表示可以被执行 read 和 write 操作的数据块大小，是 signed size_t 类型，其中 size_t 类型是在标准 C 语言库中进行定义的，在 32 位的 Linux 中 size_t 为 unsigned int 类型，即为 32 位无符号整数，在 64 位的 Linux 中其为 unsigned long，即为 64 位无符号整数。

1. read 函数的 fd 参数说明

fd 参数是待读出文件的文件描述符，其通常通过 open、create 等函数获得。

2. read 函数的 buf 参数说明

用于存放读出数据的缓冲区指针。

3. read 函数的 count 参数说明

count 是待读取的数据长度，如果 count 为 0，则 read 函数返回 0 并且没有其他结果。如果 count 大于 32767，则结果不能确定。


注 意

在 32 位系统中，count 是一个 32 位的变量，而在 64 位系统中这是一个 64 位的变量。

【例 3.13】从文件读取数据的应用实例

例 3.13 是 read 函数的应用实例，应用代码首先打开参数字符串 1 指定的文件作为源文件，然后打开参数字符串 2 指定的文件作为目的文件，最后调用 read 函数从源文件读出数据，写入到目的文件中，其流程如图 3.13 所示。

实例的应用代码如下：

```
1    //这是一个使用 read 函数把目标文件中数据读出写入到另外一个文件中的实例
2    //待读出数据文件由 argv1 参数给出，待写入数据文件由 argv2 给出
3    #include <fcntl.h>
4    #include <unistd.h>
```

```
 5  #include <stdio.h>
 6  #define PERMS 0666
 7  #define     DUMMY 0
 8  #define MAXSIZE 1024                      //常数定义
 9  int main(int argc, char *argv[])
10  {
11     int sourcefd, targetfd;                //目标文件和源文件的描述符
12     int readCounter = 0;                   //读出的字符计数器
13     char WRBuf[MAXSIZE];                   //读写缓冲区
14     if(argc!=3)                            //如果命令行参数不正确
15     {
16        printf("Plz input the correct filename as './exam311ReadFun filename1 filename2'\n");
17        return 1;
18     }
19     if((sourcefd = open(*(argv+1),O_RDONLY,DUMMY))==-1)      //如果源文件打开失败
20     {
21        printf("Source file open error!\n");
22        return 2;
23     }
24     if((targetfd = open(*(argv+2), O_WRONLY|O_CREATE, PERMS))==-1)
       //如果目标文件打开失败
25     {
26        printf("Target file open error!\n");
27        return 3;
28     }
29     while(( readCounter = read(sourcefd, WRBuf, MAXSIZE))>0)    //如果读出来的数据大于 0
30     {
31        if(write(targetfd, WRBuf,readCounter) != readCounter)     //如果写入的数据和读出的数据不同
32        {
33           printf("Target file write   error!\n");             //写数据错误
34           return 4;
35        }
36     }
37     close(sourcefd);                                        //关闭源文件
38     close(targetfd);                                        //关闭目标文件
39     return 0;
40  }
```

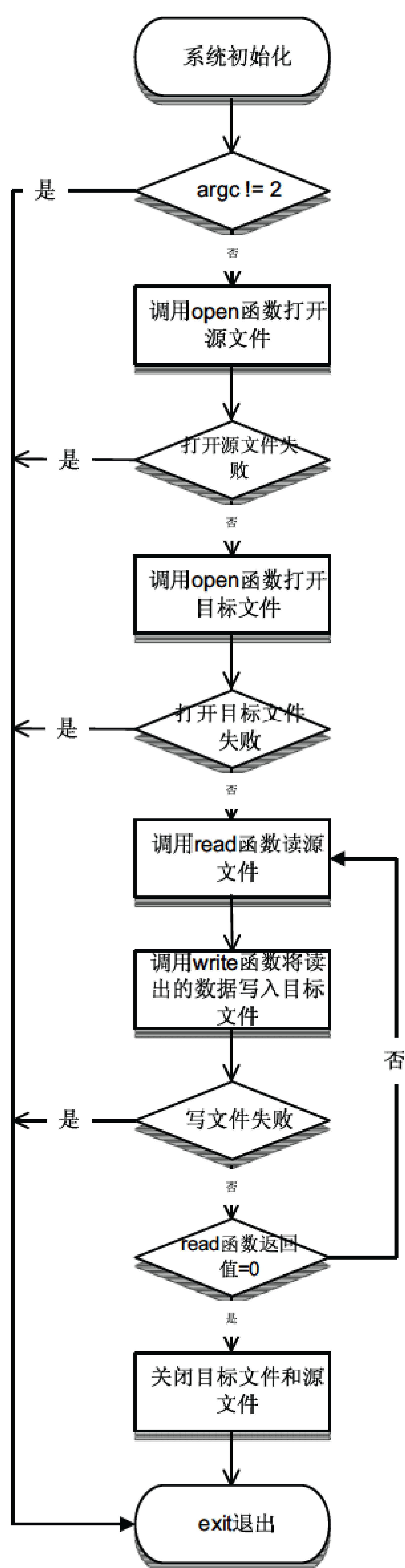


图 3.13　从文件中读取数据

在终端中使用 gcc 编译并且运行，选择例 3.12 生成的文件 lseektest 作为读操作的目标文件，将其传递给*(argv+1)参数，并且使用 readtest 作为将读出数据写入的目标文件名传递给*(argv+2)参数，执行完成之后可以使用“cat –n”命令来查看 readtest 文件的内容。

3.3　文件基础操作的综合应用——定时创建文件并且写入数据

第 3.2 节介绍了在 Linux 中如何对文件进行各种基础操作，本节是 Linux 下文件的综合应用实例。

3.3.1　综合应用的需求说明和分析

某个应用需要每隔一分钟在当前目录下建立一个以包括当前时间的字符串为文件名的文件，并且每隔一秒钟将一个当前的时间信息字符串写入文件，其流程如图 3.14 所示。

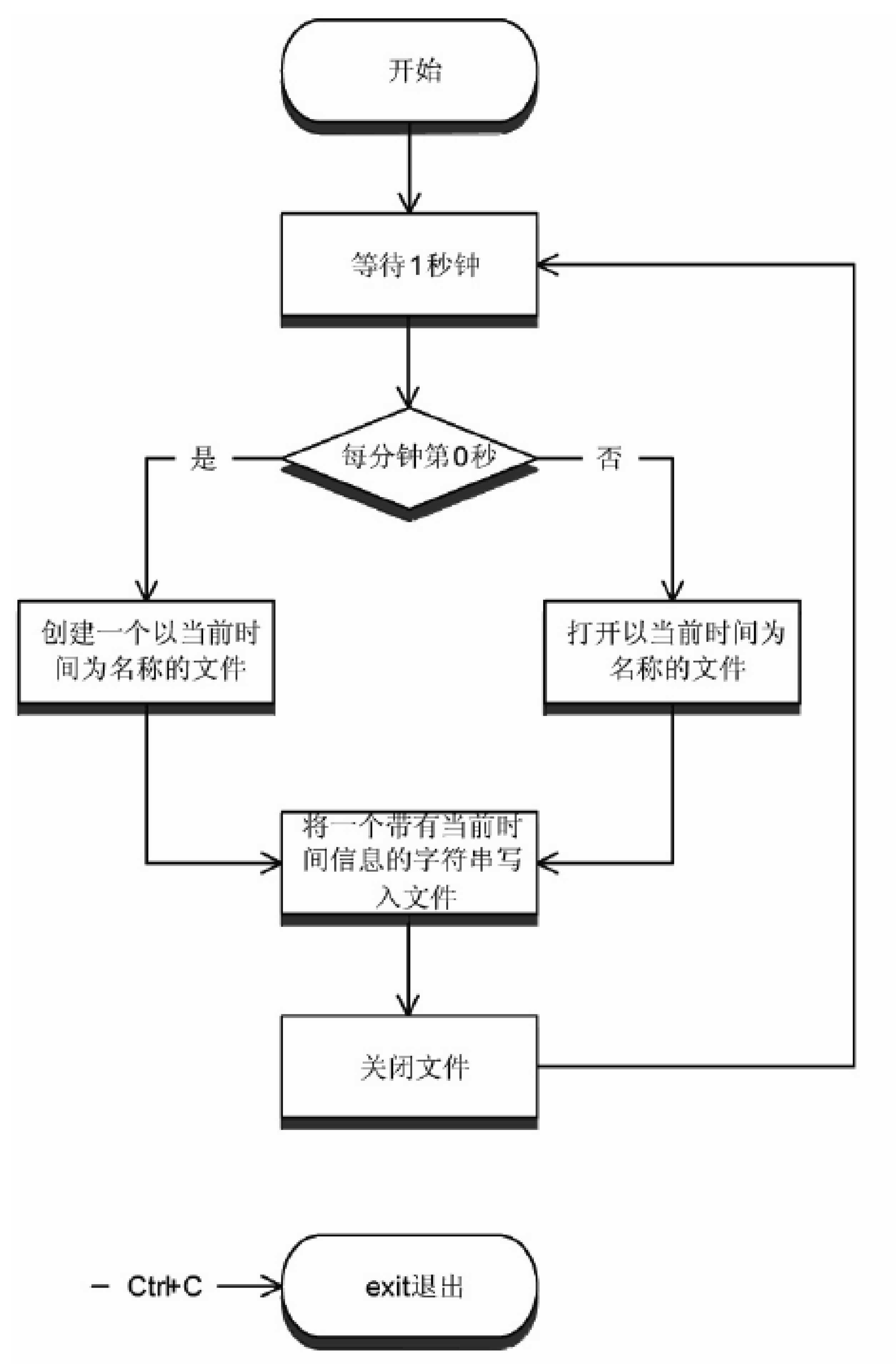


图 3.14　文件基础操作的综合应用

对应用进行分析可以得知需要考虑如下方面的实现：

- 1 秒钟的定时。
- 将当前时间信息存放进字符串，然后写入文件。
- 将当前时间信息存放进字符串，并且使用该字符串来创建一个文件。
- 将以上的各个模块综合起来。

3.3.2　秒定时的实现

秒定时是为了实现每隔一秒进行一定的操作，通常来说秒定时有两种实现方法：调用 sleep 函数或者通过对当前时间的反复查询来计算时间的差值以确定秒信息的改变。

1. 调用 sleep 函数实现秒定时

sleep 函数用于将系统挂起一段时间，以达到延迟的效果，对其标准调用格式说明如下：

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

其参数 seconds 是一个 unsigned int 类型的变量，用于传递需要将系统挂起多少秒；其返回值也是一个 unsigned int 类型的变量，如果 sleep 函数完成了需要挂起的操作，则返回值为 0，如果 sleep 函数在挂起的时候被其他操作所中断，则其返回值为已经完成了的挂起秒时间。

【例 3.14】使用 sleep 实现秒定时

例 3.14 是使用 sleep 函数来实现秒定时的实例，每隔 1 秒调用 ctime 函数在屏幕上打印当前时间，其流程如图 3.15 所示。

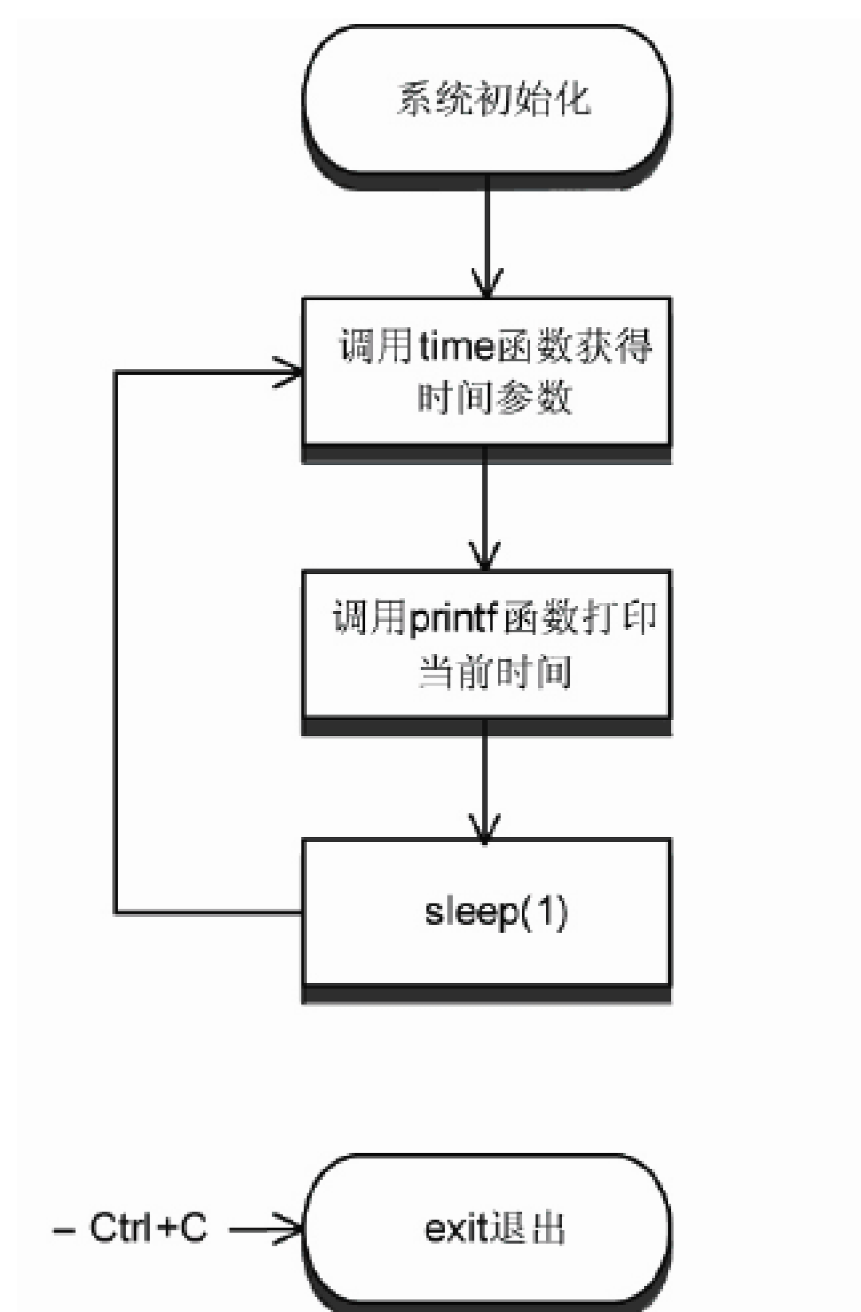


图 3.15　使用 sleep 函数实现秒定时

实例的应用代码如下：

```
1    //连续每隔 1 秒打印系统的当前时钟
2    //使用 sleep 来进行不精确延时
3    #include <time.h>
4    #include <stdio.h>
5    int main(void)
6    {
7            time_t timetemp;                    //定义一个时间结构体变量
8            while(1)
9            {
10             time(&timetemp);                  //获得时间参数
11             printf("%s",ctime(&timetemp));    //打印当前时间
12             sleep(1);
13           }
14           return 0;
15   }
```

在终端中使用 gcc 进行编译并且运行，可以看到如下输出：

```
alloy@ubuntu:~/linuxc/chapter3$ gcc exam312ConTime.c -o exam312ConTimeFun
//编译生成可执行文件
alloy@ubuntu:~/linuxc/chapter3$ ./exam312ConTimeFun
Thu Feb 21 18:09:02 2013
Thu Feb 21 18:09:03 2013
```

```
Thu Feb 21 18:09:04 2013
Thu Feb 21 18:09:05 2013
//运行，每隔 1 秒输出当前时间
^C
//使用 Ctrl+C 键中断当前执行
```



使用 sleep 函数进行的定时有两个缺点：不太精确以及在定时挂起操作过程中，如果想进行其他操作，需要终止挂起。

2. 查询时间实现秒定时

除了调用 sleep 函数之外，还可以通过对当前系统时间信息的获取和判断实现秒定时，其原理是利用 gettimeofday 函数获取当前的时间，然后和上一次获取的时间进行比较，如果相差超过 1 秒，则表明定时到达。由于应用一直在调用 gettimeofday 函数，所以其误差基本上只是 gettimeofday 函数以及计算时间差的执行时间，比较准确。

【例 3.15】查询时间实现秒定时

例 3.15 是使用 gettimeofday 函数来实现间隔 1 秒输出当前时间的实例，其流程如图 3.16 所示。

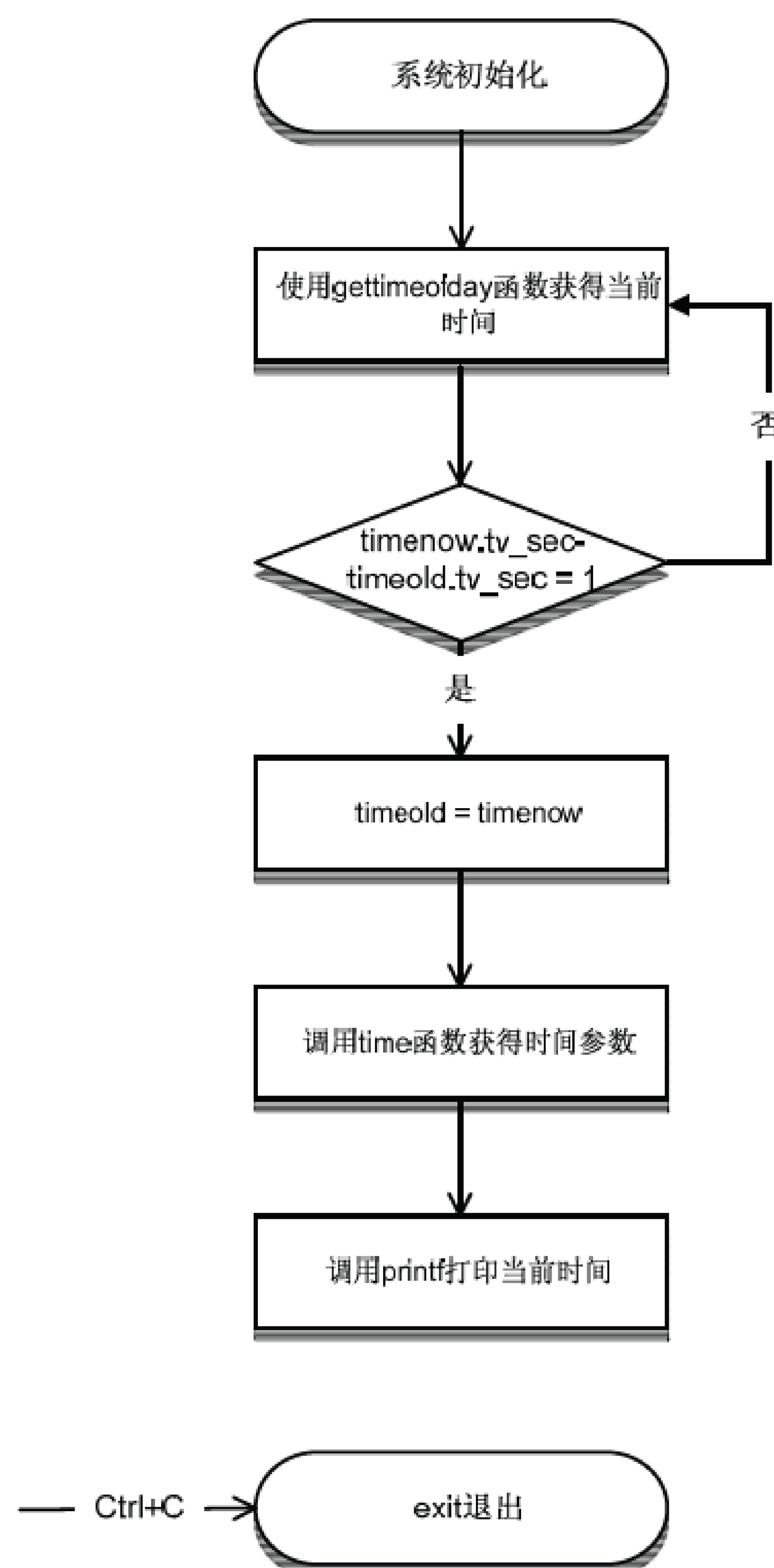


图 3.16　查询时间实现秒定时



关于 gettimeofday 函数的详细说明可以参考第 2 章的第 2.6.6 小节。

实例的应用代码如下：

```
1    //这是一个低效率的使用 gettimeofday 来获得秒定时的应用
2    //使用 gettimeofday 在 while 循环中连续获得当前的 timez 信息
3    //和之前的时间信息进行比较，如果还没到 1 秒，则等待，否则
4    //使用 break 跳出 while 循环并且打印当前时间，实现每秒打印一次
5    #include<sys/time.h>
6    #include<stdio.h>
7    int main(void)
8    {
9       struct timeval timenow,timeold;
10      struct timezone timez;
11      time_t timetemp;                                //时间结构体变量
12      gettimeofday(&timeold,&timez);                  //取得一个时间信息作为以前的数据
13      while(1)
14      {
15         while(1)
16         {
17          gettimeofday(&timenow,&timez);              //获得当前时间数据
18               if((timenow.tv_sec - timeold.tv_sec) == 1)  //如果时间过了一秒
19               {
20                  timeold = timenow;                  //更新以前的时间参考数据
21                  break;                              //退出当前循环
22               }
23          }
24          //如果还没到 1 秒，则一直等待
25          time(&timetemp);                            //获得时间参数
26          printf("%s",ctime(&timetemp));              //打印当前时间
27      }
28      return 0;
29   }
```

在终端中编译并且运行，可以看到和例 3.14 类似的输出。

3.3.3 将当前时间信息写入文件

当前的时间信息可以通过 ctime 函数获得，但是 ctime 函数的返回值是一个字符串指针，需要将其规格化之后放入缓冲区中，此时可以调用 sprintf 函数来完成。


注 意

sprintf 是一个将输入参数规格化之后存放到数组缓冲区的函数，其详细使用方法可以参考第 2.8 节。

【例 3.16】每隔 1 秒将时间信息写入文件

例 3.16 是一个每隔 1 秒获得当前时间信息然后写入指定文件的实例，其流程如图 3.17 所示。实例的应用代码如下：

```
1    //这是一个在参数指定文件中连续写入当前时间的应用
2    //文件以 1 秒为时间间隔，将当前的时间写入文件，然后回车换行
3     //这是一个使用 lseek 在一个文件中连续写入字符串的应用
```

```
4    #include <fcntl.h>
5    #include <stdio.h>
6    #include <string.h>
7    #include <sys/time.h>
8    int main(int argc,char *argv[])
9    {
10       int temp,seektemp;                              //偏移量计算中间量
11       int fd;                                         //文件描述符
12       char writebuf[50];                              //写字符串缓冲区
13       struct timeval timenow,timeold;                 //时间变量
14       struct timezone timez;
15       time_t timetemp;                                //时间结构体变量
16       int j = 0;
17       int writeCounter = 0;                           //写入计数器
18       gettimeofday(&timeold,&timez);                  //取得一个时间信息作为参考时间信息
19       if(argc!= 2)                                    //如果参数错误
20       {
21           printf("Plz input the corrcet file name as './exam39lseekFun filename string'!\n");
22           return 1;                                   //如果参数不正确则退出
23       }
24       fd = open(*(argv+1),O_RDWR|O_CREATE,S_IRWXU);
                                                         //打开文件，如果没有则创建
25       while(1)                                        //进入主循环
26       {
27          while(1)                                     //1 毫秒延时判断
28          {
29             gettimeofday(&timenow,&timez);            //获取当前时间参数
30             if((timenow.tv_sec - timeold.tv_sec) == 1)  //如果到达一秒
31             {
32                 timeold = timenow;                    //更新保存的时间信息
33                 break;                                //1 秒时间到，退出
34             }
35          }
36          time(&timetemp);                             //获得当前时间参数
37          sprintf(writebuf,"%s",ctime(&timetemp));     //将当前时间参数放入写缓冲区
38          printf("%s",&writebuf);                      //在屏幕上打印 writebuf 的内容
39          if(writeCounter == 0)                        //第一次写入
40          {
41             temp = write(fd,writebuf,strlen(writebuf));  //写入数据
42             seektemp = lseek(fd,0,SEEK_CUR);          //获得当前的偏移量
43             writeCounter++;                           //写入计数器++
44          }
45          else
46          {
47             j = strlen(writebuf) * writeCounter;      //获得偏移量
48             seektemp = lseek(fd,j,SEEK_SET);
49             temp = write(fd,writebuf,strlen(writebuf));
50             writeCounter++;
51        }
52       }
53      close(fd);
54      return 0;
55   }
```

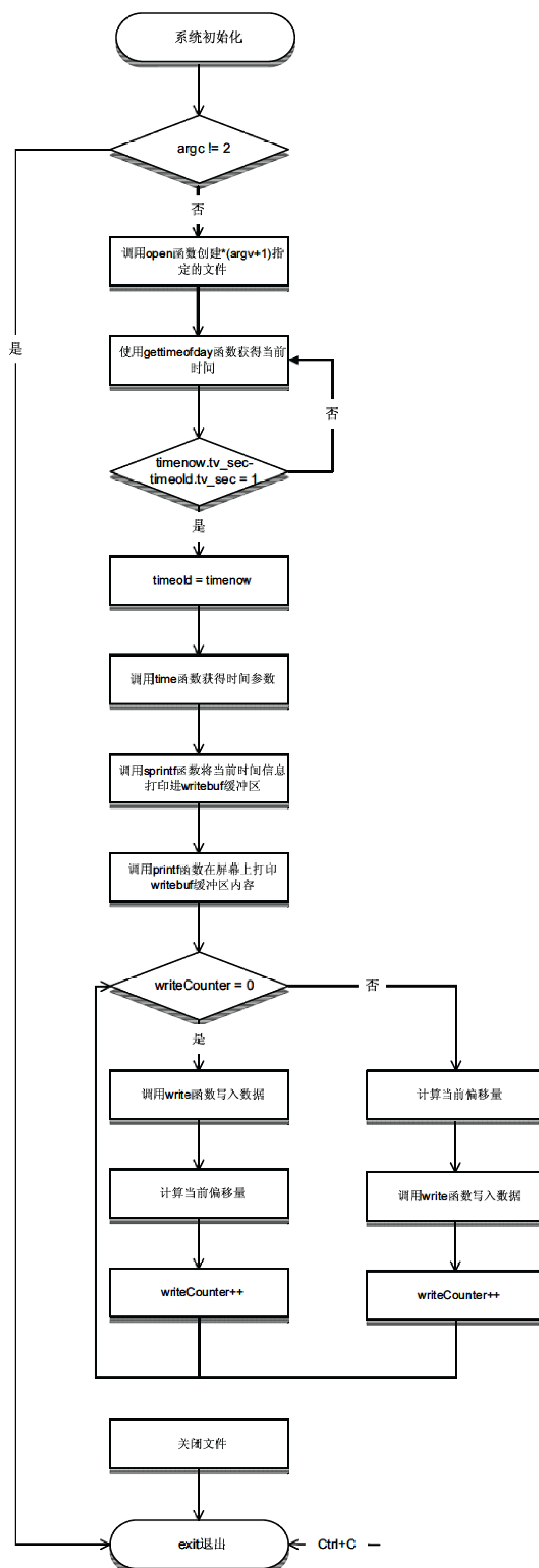


图 3.17　每隔 1 秒将当前时间写入文件

在终端中使用 gcc 进行编译并且运行，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter3$ gcc exam314ConWriteTime.c -o exam314ConWriteTimeFun
//编译链接生成可执行文件
alloy@ubuntu:~/linuxc/chapter3$ ./exam314ConWriteTimeFun conwritetest
Thu Feb 21 20:09:25 2013
Thu Feb 21 20:09:26 2013
Thu Feb 21 20:09:27 2013
Thu Feb 21 20:09:28 2013
Thu Feb 21 20:09:29 2013
//每隔 1 秒将时间信息字符串写入文件 conwritetest，并且将当前时间信息显示在屏幕上
^C
//使用 Ctrl+C 中断当前程序运行
alloy@ubuntu:~/linuxc/chapter3$ cat -n conwritetest
 1      Thu Feb 21 20:09:25 2013
 2      Thu Feb 21 20:09:26 2013
 3      Thu Feb 21 20:09:27 2013
 4      Thu Feb 21 20:09:28 2013
 5      Thu Feb 21 20:09:29 2013
//使用“cat –n”命令查看 conwritetest 文件中的内容
```

3.3.4 使用时间信息作为文件名

使用时间信息作为文件名的时候需要首先将时间信息分离出来，组成一个“时+分+秒”的字符串，然后传递给对应的函数以创建文件，此时可以调用 time 函数来获得当前时间，其返回的时、分、秒信息会分别被存储到结构体的 hour、min 和 sec 分量中。


注 意

time 函数的详细使用方法可以参考第 2 章的第 2.7 节。

【例 3.17】使用当前时间信息作为文件名来创建文件

例 3.17 是一个使用当前时间作为文件名创建一个文件并且将创建时的时间信息写入文件的实例，其文件名的结构是“File+时+分+秒”，其流程如图 3.18 所示。



图 3.18　使用时间信息作为文件名创建文件

实例的应用代码如下：

```
1    //这是一个利用当前时间作为参数来创建新文件的应用
2    //新文件的格式为 File+时+分+秒
3    //应用代码首先使用 time 系列函数获得当前的时、分、秒信息
4    //然后通过组合获得对应的字符串，传递给 Open 函数创建文件
5    //最后在文件中写入一个含有时间参数的字符串
6    #include <time.h>
7    #include <stdio.h>
8    #include <string.h>
9    #include <fcntl.h>
10   int main(void)
11   {
12           time_t timetemp;                    //定义一个时间结构体变量
13           struct tm *p;                       //结构体指针
14           int i;
```

```
15          char timebuf[7];                              //时间信息，注意加上“\0”
16          char writetimebuf[7];                         //写文件时间缓冲区
17          char filenamebuf[10] = "File";                //文件头
18          char writebuf[30]="this is a test! the time is ";
19          char enterbuf[3]="\r\n";                      //回车换行 buf
20          int fd;
21          int temp;
22          time(&timetemp);                              //获得时间参数
23          printf("当前时间为%s",asctime(gmtime(&timetemp)));
                                                          //不需要添加回车换行符
24          p = localtime(&timetemp);
25          printf("%d:%d:%d\n",p->tm_hour,p->tm_min,p->tm_sec);
26          sprintf(timebuf,"%02d%02d%02d",p->tm_hour,p->tm_min,p->tm_sec);
27          //将时、分、秒信息按照 2 位前端补 0 的方式格式化送入时间 buf
28          printf("step1 timebuf is %s\n",timebuf);
29          strcpy(writetimebuf,timebuf);                           //复制字符串
30          printf("writetimebuf is %s\n",writetimebuf);
31          strcat(filenamebuf,timebuf);
32          printf("step2 timebuf is %s\n",timebuf);
33          printf("filenamebuf is %s\n",filenamebuf);
34          fd = open(filenamebuf,O_RDWR|O_CREATE,S_IRWXU);         //创建文件
35          strcat(writebuf,writetimebuf);                          //连接两个字符串
36          strcat(writebuf,enterbuf);                              //回车换行
37          temp = write(fd,writebuf,strlen(writebuf));       //写入一个字符串以表示正确
38          temp = close(fd);
39          return 0;
40      }
```

在终端中使用 gcc 对其进行编译并且运行，可以看到如下输出：

```
alloy@ubuntu:~/linuxc/chapter3$ gcc exam316timeOpen.c -o exam316timeOpenFun
//编译链接生成可执行文件
alloy@ubuntu:~/linuxc/chapter3$ ./exam316timeOpenFun
//输出一些调试信息
当前时间为 Fri Feb 22 09:36:36 2013
17:36:36
step1 timebuf is 173636
writetimebuf is 173636
step2 timebuf is
filenamebuf is File173636
//以上为文件名缓冲区的数据
alloy@ubuntu:~/linuxc/chapter3$ cat -n File173636
this is a test! the time is 173636
//使用“cat –n”命令获得建立文件的内容
```

【例 3.18】实例的综合

将前面的内容进行综合，即可得到符合需求的应用，如例 3.18 所示，每隔 1 分钟创建一个文件，并且每隔 1 秒将当前时间信息写入到文件中，其流程如图 3.19 所示。

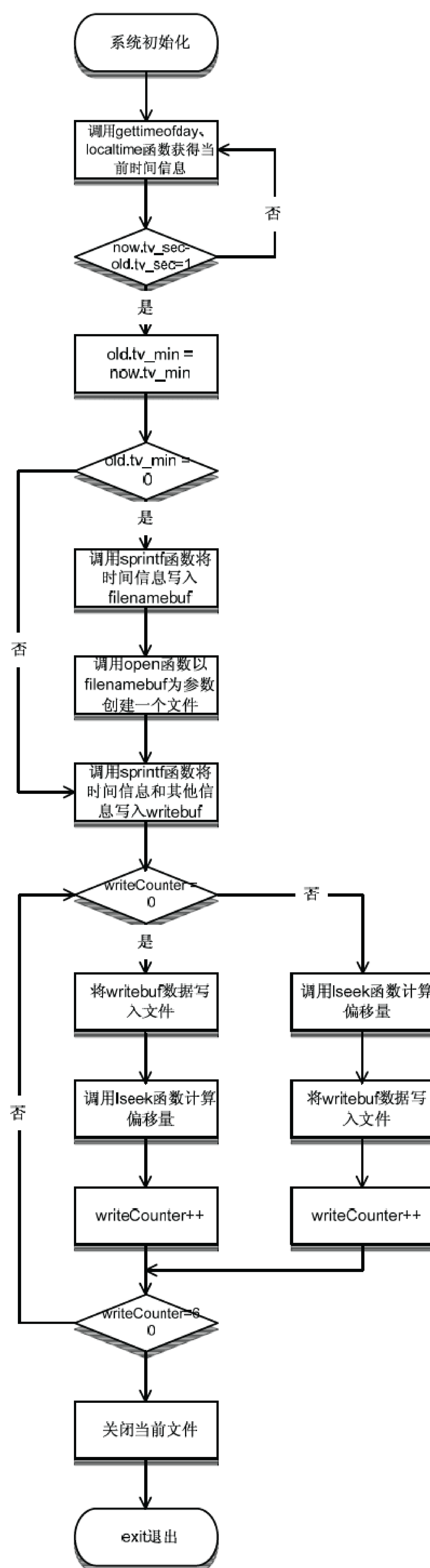


图 3.19　文件基础操作综合应用

实例的应用代码如下：

```
1    //这是一个在参数指定文件中连续写入当前时间的应用
2    //每隔 1 分钟在当前目录下建立一个新文件，通过对 tm_sec 是否为 0 来判断
3    //文件以 1 秒为时间间隔，将当前的时间写入文件，然后回车换行
4    //这是一个使用 lseek 在一个文件中连续写入字符串的应用
5    #include <fcntl.h>
6    #include <stdio.h>
7    #include <string.h>
8    #include <sys/time.h>
9    int main(int argc,char *argv[])
10   {
11     time_t filetime;
12     struct tm *p;
13     int temp,seektemp;                          //偏移量计算中间量
14     int fd;                                     //文件描述符
15     char writebuf[50];                          //写字符串缓冲区
16     char filenamebuf[10] = "File";              //文件头
17     char timebuf[7];                            //时、分、秒信息缓冲区
18     struct timeval timenow,timeold;             //时间变量
19     struct timezone timez;
20     //time_t filetime;                          //时间结构体变量
21     //struct tm *p;                             //时间结构体指针
22     int j = 0;
23     int writeCounter = 0;                       //写入计数器
24     gettimeofday(&timeold,&timez);              //取得一个时间信息作为参考时间信息
25     if(argc!= 2)                                //如果参数错误
26     {
27         printf("Plz input the corrcet file name as './exam39lseekFun filename string'!\n");
28         return 1;                               //如果参数不正确则退出
29     }
30     fd = open(*(argv+1),O_RDWR|O_CREATE,S_IRWXU);  //打开文件，如果没有则创建
31     while(1)                                    //进入主循环
32     {
33       while(1)                                  //1 毫秒延时判断
34       {
35         gettimeofday(&timenow,&timez);          //获取当前时间参数
36         time(&filetime);
37         p = localtime(&filetime);               //获得时、分、秒参数，以供创建新文件
38         sprintf(timebuf,"%02d%02d%02d",p->tm_hour,p->tm_min,p->tm_sec);
39         printf("%d:%d:%d\n",p->tm_hour,p->tm_min,p->tm_sec);
40         //      sprintf(timebuf,"%02d%02d%02d",p->tm_hour,p->tm_min,p->tm_sec);
41         //时分秒信息放入 timebuf 缓冲区备用
42         gettimeofday(&timenow,&timez);          //获取当前时间参数
43         if((timenow.tv_sec - timeold.tv_sec) == 1)   //如果到达一秒
44         {
45             timeold = timenow;                  //更新保存的时间信息
46             break;                              //1 秒时间到，退出
47         }
48       }
49       if(timeold.tv_sec == 0)                   //如果是 0 秒
```

```
50              {
51                  strcat(filenamebuf,timebuf);                        //创建文件名
52                  fd = open(filenamebuf,O_RDWR|O_CREATE,S_IRWXU); //创建文件
53              }
54              time(&timetemp);            //获得当前时间参数
55              sprintf(writebuf,"%s",ctime(&timetemp));            //将当前时间参数放入写缓冲区
56              printf("%s",&writebuf);                             //在屏幕上打印 writebuf 的内容
57              if(writeCounter == 0)                               //第一次写入
58              {
59                  temp = write(fd,writebuf,strlen(writebuf));     //写入数据
60                  seektemp = lseek(fd,0,SEEK_CUR);                //获得当前的偏移量
61                  writeCounter++;                                 //写入计数器++
62              }
63              else
64              {
65                  j = strlen(writebuf) * writeCounter;            //获得偏移量
66                  seektemp = lseek(fd,j,SEEK_SET);
67                  temp = write(fd,writebuf,strlen(writebuf));
68                  writeCounter++;
69             }
70          }
71       close(fd);
72       return 0;
73   }
```

在终端中对其进行编译运行，即可看到对应的输出。

3.4 本章习题

1.使用 open 函数来编写一个程序，打开或者创建一个指定的文件，带路径文件名由用户指定输入，文件名的最长长度为 30。

2.编写一个程序，用于测试标准输入文件（文件描述符 0）是否能使用 lseek 函数来设置位移量。

3.使用 write 函数来编写一个程序，在程序中指定一个文件，用户可以向程序中一次写入不超过 80 个字符的数据。

4.使用 read 和 write 函数，编写一个程序，实现 cp 函数的基础功能。

5.当使用 lseek 函数定位到超出文件尾端之后，对于新写入的数据需要分配磁盘块，但是对于原文件尾端和新开始写位置之间的部分不需要分配磁盘块，这会产生空洞文件，文件中的空洞并不要求在磁盘上占有存储区，具体的处理方式与文件系统的实现有关，请尝试使用 open、lseek 等函数创建一个含有空洞的文件。

第 4 章　Linux 的目录文件操作

目录文件是 Linux 文件的一种，在实际应用中通常用于存放一组文件（这些文件中可以包括其他目录文件），本章将介绍对目录文件进行操作的方法，涉及如下的内容：

- 在 Linux 中创建和删除一个目录。
- 在 Linux 中对一个目录进行打开、关闭和读取操作。
- 修改当前的工作目录路径。

4.1　目录文件的基础操作

Linux 目录文件的基础操作包括目录文件的创建、删除、打开、关闭、读取以及修改当前工作目录路径。

4.1.1　目录文件的创建和删除

1. 创建目录文件

mkdir 函数用于在文件系统中建立一个目录，其会自动在目录中创建“.”和“..”目录项，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int mkdir(const char *pathname, mode_t mode);
```

其中 pathname 为目录的带路径名称，mode 为目录的权限，其意义和普通文件相同，所以可以参考第 3 章的表 3.3，需要注意的是对于目录来说最少要设置一个执行权限位以允许用户访问该目录中的文件。

这个新创建目录的用户 ID 被设置为调用进程的有效用户 ID，其组 ID 则为父目录的组 ID 或者进程的有效组 ID。在新建一个目录之后，mkdir 将更新该目录的 st_atime、st_ctime 和 st_mtime，同时更新其父目录的 st_ctime 和 st_mtime。


注　意

由 pathname 指定的新目录的父目录必须存在，并且调用进程必须具有该父目录的写权限以及 pathname 涉及的各个分路径目录的搜寻权限。

【例 4.1】在 Linux 中创建一个指定目录

例 4.1 是一个使用 mkdir 函数来创建一个目录文件的实例，目录的名称由参数 argv 给出，其流程如图 4.1 所示。

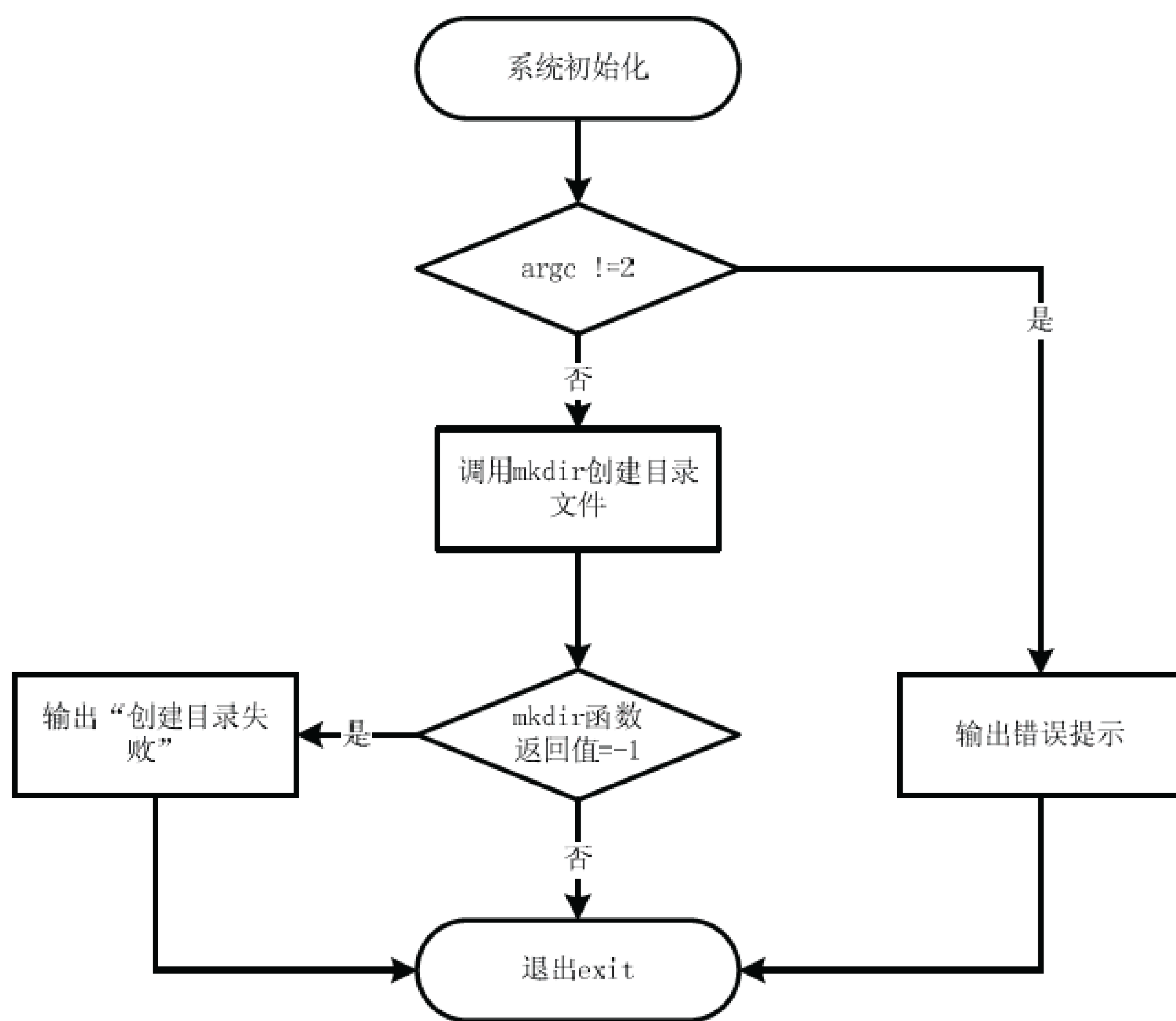


图 4.1　使用 mkdir 函数来创建一个目录

实例的应用代码如下：

```
1    //这是一个使用 mkdir 来创建目录的应用实例
2    //目录的名称由 argv 给出
3    #include <fcntl.h>
4    #include <stdio.h>
5    int main(int argc,char *argv[])
6    {
7        int temp;
8        if(argc!=2)                    //如果参数格式不正确
9        {
10           printf("文件参数错误!\n");
11           return 1;                  //退出
12       }
13       temp = mkdir(*(argv+1),S_IRWXU|S_IRGRP|S_IXOTH); //必须最少指定一个执行权限位
14       if(temp == -1)                 //如果创建目录失败
15       {
16           printf("创建目录失败\n");
17           return 2;                  //退出
18       }
19       return 0;
20   }
```

在终端中使用 gcc 对其进行编译并且运行，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter4$ gcc exam401mkdir.c -o exam401mkdir
//编译链接生成可执行文件 exam401mkdir
alloy@ubuntu:~/linuxc/chapter4$ ./exam401mkdir testdir
//调用可执行文件创建目录 testdir
alloy@ubuntu:~/linuxc/chapter4$ ls
//使用 ls 命令查看 testdir 文件是否已经被创建
exam401mkdir    exam401mkdir.c    testdir
```

在 Linux 的 GNOME 图形界面下可以看到如图 4.2 所示的三个文件，分别是刚刚被创建的目录文件、可执行文件 exam401mkdir 以及 C 语言源文件 exam401mkdir.c。

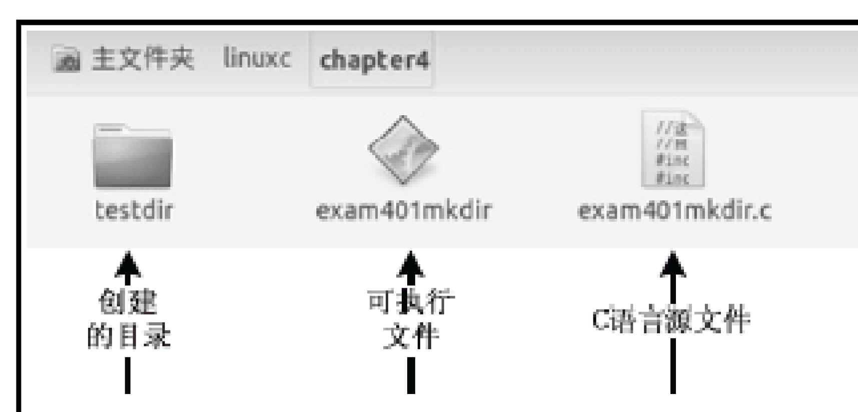


图 4.2　创建的目录文件

同时可以查看该目录文件 testdir 的属性，如图 4.3 所示。



图 4.3　目录文件 testdir 的属性

目录文件的权限如图 4.4 所示，这是由“S_IRWXU|S_IRGRP|S_IXOTH”参数所决定的。



图 4.4　目录文件 testdir 的权限

2. 删除目录文件

rmdir 函数用于在文件系统中删除一个目录，但是这个目录必须是空目录（只包括“.”和“..”文件项），对其标准调用格式说明如下，如果调用成功则返回 0，否则返回-1。

```
#include <unistd.h>
int rmdir(const char *pathname);
```

其中 pathname 为目录的带路径名称。

需要注意的是如果此调用使目录的连接计数为 0，并且也没有其他进程打开此目录，则释放由此目录占用的空间。如果在连接计数达到 0 时，有一个或几个进程打开了此目录，则在此函数返回前删除最后一个连接。另外，在此目录中不能再创建新文件。但是在最后一个进程关闭它之前并不释放此目录（即使某些进程打开该目录，它们在此目录下也不能执行其他操作，因为为使 rmdir 函数成功执行，该目录必须是空的）。

【例 4.2】在 Linux 中删除一个指定目录

例 4.2 是一个使用 rmdir 函数来删除一个指定目录文件的实例，目录的名称由参数 argv 给出，其流程如图 4.5 所示。

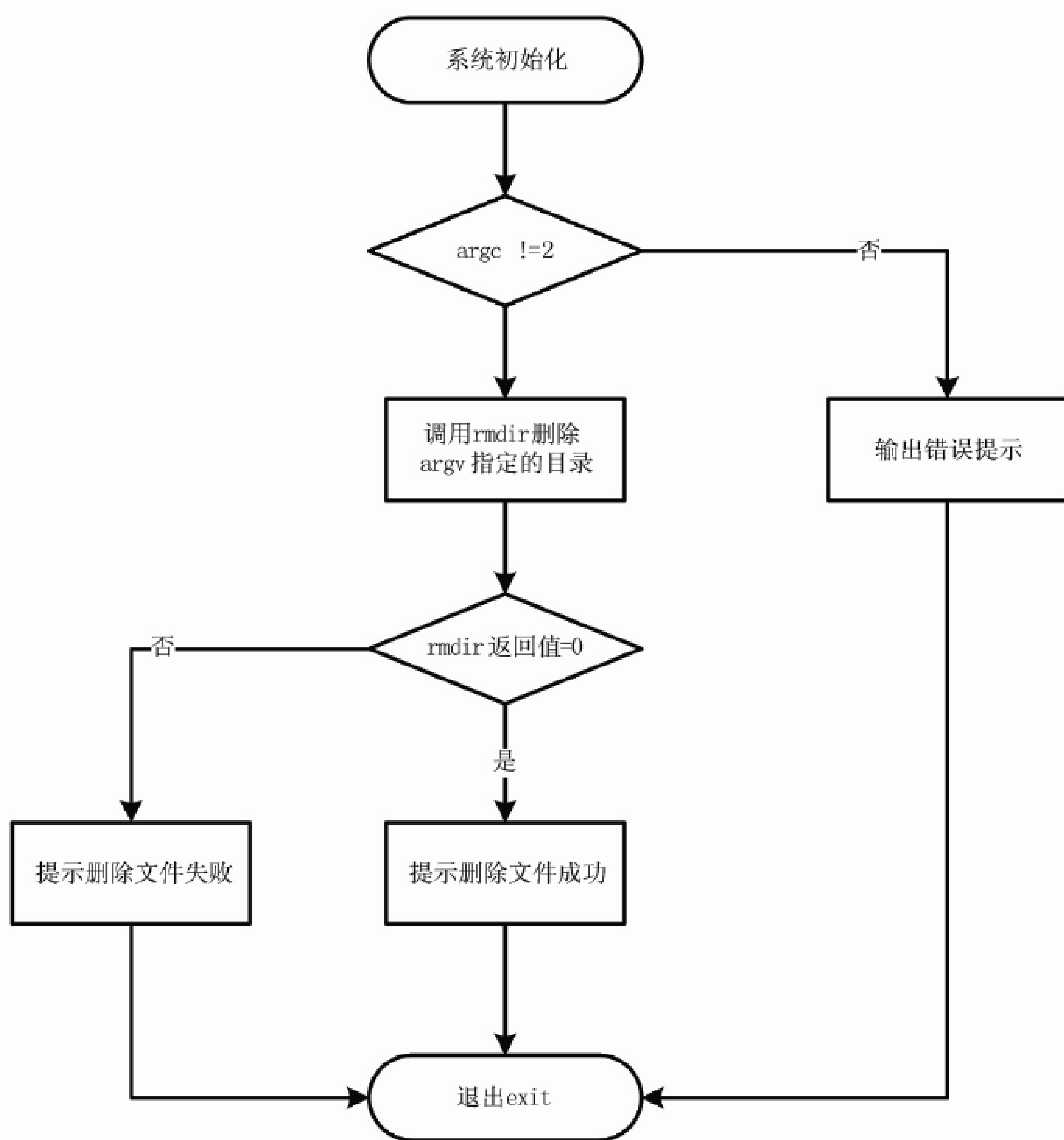


图 4.5　使用 rmdir 来删除一个目录

实例的应用代码如下：

```
1    //使用 rmdir 函数删除一个目录
```

```
2    //目录的名称由 argv 传递
3    #include <fcntl.h>
4    #include <stdio.h>
5    int main(int argc,char *argv[])
6    {
7       int temp;
8       if(argc != 2)                                    //如果参数错误
9       {
10         printf("请输入正确的参数!\n");
11         return 1;                                     //参数错误，退出
12      }
13      temp = rmdir(*(argv+1));                         //删除目录文件
14      if(temp == 0)
15      {
16          printf("删除目录%s 成功\n",*(argv+1));          //删除完成
17      }
18      else
19      {
20              printf("删除目录%s 失败\n",*(argv+1));      //删除失败
21      }
22      return 0;
23   }
```

在终端中使用 gcc 对其进行编译并且运行，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter4$ gcc exam402rmdir.c -o exam402rmdir
//编译链接生成可执行文件 exam402rmdir
alloy@ubuntu:~/linuxc/chapter4$ ls
exam401mkdir   exam401mkdir.c   exam402rmdir   exam402rmdir.c   testdir
//使用 ls 命令查看当前目录下的各个文件
alloy@ubuntu:~/linuxc/chapter4$ ./exam402rmdir testdir
删除目录 testdir 成功
//调用 exam402rmdir 可执行文件对 testdir 进行删除，返回删除成功信息
alloy@ubuntu:~/linuxc/chapter4$ ls
exam401mkdir   exam401mkdir.c   exam402rmdir   exam402rmdir.c
//再次调用 ls 命令查看当前目录，testdir 目录文件已经不见了
```

【例 4.3】验证要求被删除目录非空

由于 rmdir 函数要求当前的被删除目录非空，所以如果当目录非空的时候对该目录进行删除操作会失败，可以利用如例 4.3 所示的操作过程来验证这一点，其执行流程如下：

01 调用 exam401mkdir 在当前目录下创建一个 nonopdir 的目录。

02 使用 vim 在 nonopdir 目录下创建一个 nonop.txt 的文件，在其中任意输入一些字符并且保存。

03 调用 exam402rmdir 试图删除 nonopdir 目录，由于该目录非空，则删除失败。

04 使用 rm 命令删除 nonopdir 目录下的 nonop.txt 文件，然后再次调用 exam402rmdir 删除 nonopdir 目录，此时由于目录已空，则删除成功。

实例的操作步骤如下：

```
alloy@ubuntu:~/linuxc/chapter4$ ls
exam401mkdir   exam401mkdir.c   exam402rmdir   exam402rmdir.c
alloy@ubuntu:~/linuxc/chapter4$ ls
exam401mkdir   exam401mkdir.c   exam402rmdir   exam402rmdir.c
alloy@ubuntu:~/linuxc/chapter4$ ./exam401mkdir nonopdir
alloy@ubuntu:~/linuxc/chapter4$ ls
exam401mkdir   exam401mkdir.c   exam402rmdir   exam402rmdir.c   nonopdir
alloy@ubuntu:~/linuxc/chapter4$ cd nonopdir
alloy@ubuntu:~/linuxc/chapter4/nonopdir$ vim nonop.txt
alloy@ubuntu:~/linuxc/chapter4/nonopdir$ ls
nonop.txt
alloy@ubuntu:~/linuxc/chapter4/nonopdir$ cd ..
alloy@ubuntu:~/linuxc/chapter4$ ls
exam401mkdir   exam401mkdir.c   exam402rmdir   exam402rmdir.c   nonopdir
alloy@ubuntu:~/linuxc/chapter4$ ./exam402rmdir nonopdir
删除目录 nonopdir 失败
alloy@ubuntu:~/linuxc/chapter4$ rm nonopdir/nonop.txt
alloy@ubuntu:~/linuxc/chapter4$ ./exam402rmdir nonopdir
删除目录 nonopdir 成功
```

4.1.2 目录文件的打开、关闭和读取

在 Linux 系统中，对目录有访问权限的用户都可以对目录进行读操作，但是只有操作系统内核才有权限对目录进行写操作。

1. 打开和关闭目录文件

opendir 函数用于打开一个目录，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (const char *pathname);
```

函数的参数是目录的完整路径名，如果操作成功，则函数返回一个 DIR 类型的指针，如果操作失败则返回 NULL。DIR 指针是一个内部结构，其具体结构将在后续章节中进行介绍。


注 意

和前面介绍的 open 函数不同，opendir 函数并不能创建一个目录文件，所以如果该目录文件不存在，则会导致打开目录文件失败。

和文件操作相同，打开的目录在操作完成之后也必须进行关闭操作，此时可以调用 closedir 函数，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <dirent.h>
int closedir (DIR *dp);
```

其中参数 dp 是一个指向待关闭目录的 DIR 类型指针，如果操作成功，则返回 0，否则返回“-1”。

【例 4.4】在 Linux 中打开和关闭一个目录

例 4.4 是利用 opendir 函数打开一个指定目录然后关闭的实例，如果该目录不存在，则调用 mkdir 函数创建该目录。使用*dp 作为被打开目录文件的返回指针，指定目录的名称由 argv 参数给出，如果*dp 为空，则表明打开目录文件失败，调用 mkdir 函数创建该指定文件并且赋予该目录文件的属性为“S_IRWXU|S_IRGRP|S_IXOTH”，其流程如图 4.6 所示。

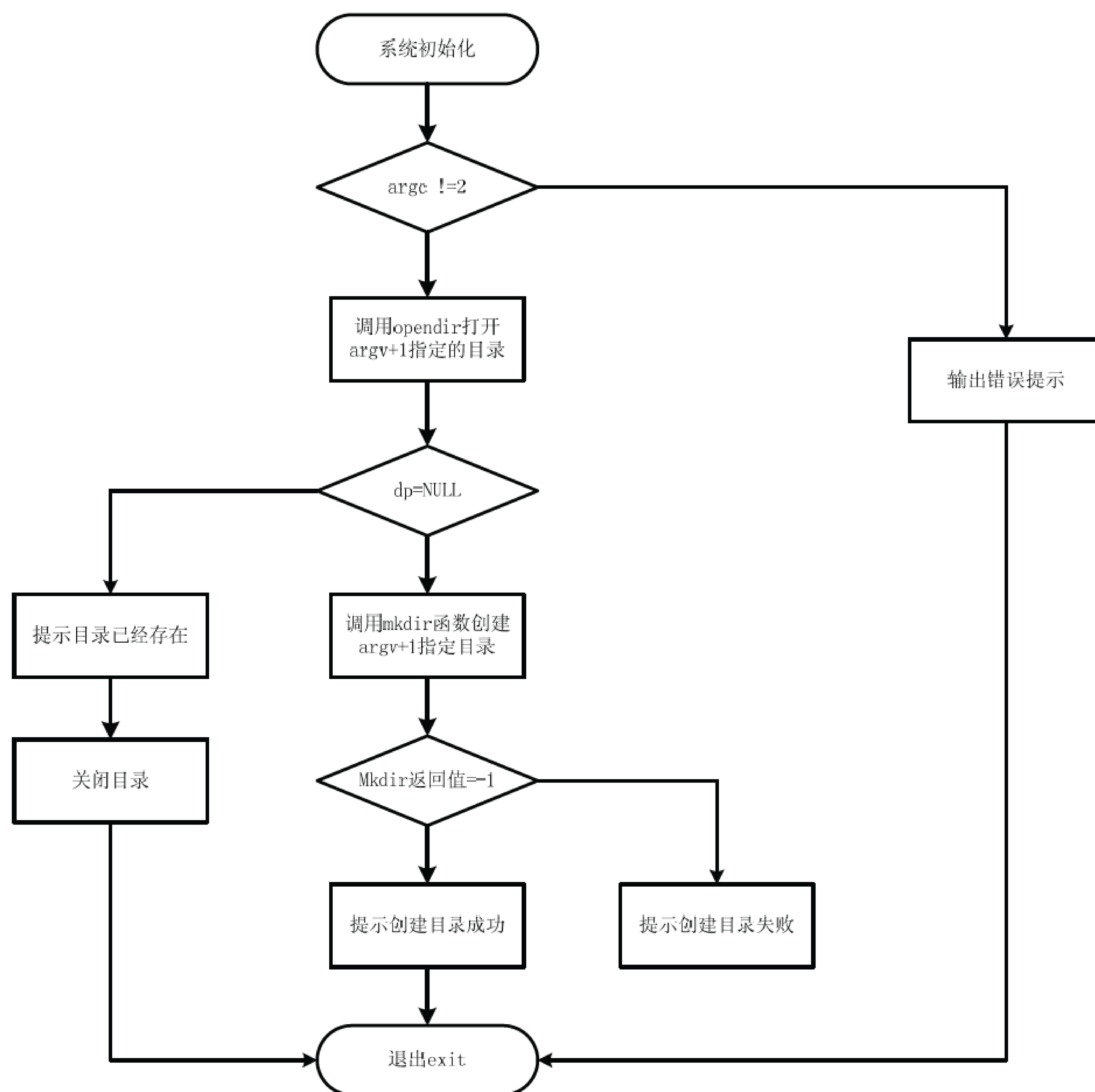


图 4.6　在 Linux 中打开和关闭一个目录

实例的应用代码如下：

```
1    //判断在当前工作路径下某个目录是否存在
2    //如果不存在则创建该目录
3    //目录名由 argv 参数传递进去
4    #include <fcntl.h>
5    #include <sys/types.h>
6    #include <dirent.h>
```

```
 7  #include <stdio.h>
 8  int main(int argc,char *argv[])
 9  {
10     DIR *dp;                                    //目录文件指针
11     int temp;                                   //存放 mkdir 函数的返回值
12     if(argc != 2)                               //如果参数不正确
13     {
14        printf("请输入正确的参数！ /n");
15        return 1;
16     }
17     dp = opendir(*(argv+1));                    //尝试打开目录
18     if(dp == NULL)                              //出错，说明目录不存在
19     {
20        printf("目录不存在！ \n");
21        temp = mkdir(*(argv + 1),S_IRWXU|S_IRGRP|S_IXOTH);          //创建目录文件
22        if(temp == -1)
23        {
24           printf("创建目录失败！ \n");
25           return 2;
26        }
27        else
28        {
29           printf("创建目录%s 成功！ \n",*(argv+1));
30        }
31     }
32     else
33     {
34         printf("目录%s 已经存在！ 打开成功！ \n",*(argv+1));
35         closedir(dp);                                               //关闭目录
36     }
37     return 0;
38  }
```

在终端中使用 gcc 对其进行编译并且运行，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter4$ gcc exam403opendir.c -o exam403opendir
//编译链接生成可执行文件
alloy@ubuntu:~/linuxc/chapter4$ ./exam403opendir opendirtest
目录不存在！
创建目录 opendirtest 成功！
//调用 exam403opendir 可执行文件试图打开 opendirtest 目录，提示目录不存在
//创建 opendirtest 目录成功
alloy@ubuntu:~/linuxc/chapter4$ ./exam403opendir opendirtest
目录 opendirtest 已经存在！ 打开成功！
//再次调用 exam403opendir 打开 opendirtest 目录，此时目录已经存在，打开成功
```

2. 读取目录文件

对目录的读取操作可以通过调用 readdir 函数来完成，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir (DIR *dp);
```

若操作成功则返回一个 dirent 类型的指针，若位于目录尾或出错则为 NULL。

参数 dp 指向要读取的目录，函数返回值为指向 dirent 结构体的指针。dirent 定义在头文件<dirent.h>中：

```
struct dirent
{
ino_t d_ino;                    /*i-node number*/
char d_name[NAME_MAX + 1];      /*null-terminated filename*/
}
```

其中 d_ino 用于表示该目录的节点号，d_name 用于存放此目录链接的文件名。当目录中没有更多链接时，其值为 0。

【例 4.5】统计目录内文件

例 4.5 是一个调用 opendir 和 readdir 函数对指定目录进行遍历操作，然后打印输出指定目录中各种类型的文件的相应数据实例，指定目录的名称通过 argv 参数给出，其流程如图 4.7 所示。

实例的应用代码如下：

```
1   //调用 opendir 和 readdir 函数对指定目录进行遍历操作
2   //然后打印输出指定目录中各种类型的文件数目
3   #include <stdio.h>
4   #include <fcntl.h>
5   #include <dirent.h>
6   #include <limits.h>
7   #include <sys/stat.h>
8   #include <string.h>
9   #include <stdlib.h>
10
11  typedef int Myfunc(const char *, const struct stat *, int);         //定义一个函数
12  static Myfunc myfunc;
13  static int myftw(char *, Myfunc *);
14  static int dopath(Myfunc *);
15  static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;
16  //各种类型的文件数目对应的变量
17  char *path_alloc(int* size);
18
19  int main(int argc, char *argv[])
20  {
21      int ret;
22      if (argc != 2)
23      {
24          printf("请输入正确的参数!\n");                             //参数错误
25          return 1;
26      }
```

```
27     ret = myftw(argv[1], myfunc);                          /* does it all */
28     ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
29     //计算文件总量
30     if (ntot == 0)        //如果目录中没有文件，则将 ntot 设置为 1 以避免除数为 0
31     {
32        ntot = 1;
33     }
34     //以下为一次打印各种类型文件的数据
35     printf("普通文件  = %7ld, %5.2f %%\n", nreg, nreg*100.0/ntot);
36     printf("目录文件  = %7ld, %5.2f %%\n", ndir,ndir*100.0/ntot);
37     printf("块设备文件  = %7ld, %5.2f %%\n", nblk,nblk*100.0/ntot);
38     printf("字设备文件  = %7ld, %5.2f %%\n", nchr, nchr*100.0/ntot);
39     printf("FIFOs = %7ld, %5.2f %%\n", nfifo,nfifo*100.0/ntot);
40     printf("符号链接文件  = %7ld, %5.2f %%\n", nslink, nslink*100.0/ntot);
41     printf("套接字文件  = %7ld, %5.2f %%\n", nsock,nsock*100.0/ntot);
42     return ret;
43  }
44  //路径缓冲区分配函数
45  char *path_alloc(int* size)
46  {
47     char *p = NULL;
48     if(!size)
49     {
50        return NULL;
51     }
52     p = malloc(256);
53     if(p)
54     {
55        *size = 256;
56     }
57     else
58     {
59        *size = 0;
60     }
61     return p;
62  }
63
64  #define FTW_F     1
65  #define FTW_D     2                      //目录
66  #define FTW_DNR 3                        //不能读的目录
67  #define FTW_NS    4                      //不能获得状态的文件
68
69  static char       *fullpath;             //存放每个文件的完整路径
70
71  static int myftw(char *pathname, Myfunc *func)
72  {
73     int len;
74     fullpath = path_alloc(&len);          //给路径缓冲区分配一个长度
75     strncpy(fullpath, pathname, len);     //复制文件名称
```

```
 76     fullpath[len-1] = 0;
 77     return(dopath(func));
 78  }
 79  //获得文件的状态
 80  static int dopath(Myfunc* func)
 81  {
 82     struct stat     statbuf;
 83     struct dirent *dirp;
 84     DIR *dp;
 85     int ret;
 86     char *ptr;
 87     if (lstat(fullpath, &statbuf) < 0)              //获得文件状态失败
 88     {
 89        return(func(fullpath, &statbuf, FTW_NS));
 90     }
 91     if (S_ISDIR(statbuf.st_mode) == 0)              //如果不是目录
 92     {
 93        return(func(fullpath, &statbuf, FTW_F));
 94     }
 95     if ((ret = func(fullpath, &statbuf, FTW_D)) != 0)
 96     {
 97        return(ret);
 98     }
 99     ptr = fullpath + strlen(fullpath);              //指向路径缓冲区结尾
100     *ptr++ = '/';
101     *ptr = 0;
102     if ((dp = opendir(fullpath)) == NULL)           //如果不能读目录
103     {
104        return(func(fullpath, &statbuf, FTW_DNR));
105     }
106     while ((dirp = readdir(dp)) != NULL) {
107                    if (strcmp(dirp->d_name, ".") == 0   ||
108                        strcmp(dirp->d_name, "..") == 0)
109                               continue;                 /* ignore dot and dot-dot */
110                    strcpy(ptr, dirp->d_name);           /* append name after slash */
111                    if ((ret = dopath(func)) != 0)       /* recursive */
112                           break;   /* time to leave */
113           }
114           ptr[-1] = 0;     /* erase everything from slash onwards */
115
116           if (closedir(dp) < 0)
117           {
118                    printf("can't close directory %s\n", fullpath);
119        }
120           return(ret);
121  }
122
123  static int myfunc(const char *pathname, const struct stat *statptr, int type)
124  {
```

```
125         switch (type) {
126         case FTW_F:
127                 switch (statptr->st_mode & S_IFMT) {
128                 case S_IFREG:    nreg++;          break;
129                 case S_IFBLK:    nblk++;          break;
130                 case S_IFCHR:    nchr++;          break;
131                 case S_IFIFO:    nfifo++;         break;
132                 case S_IFLNK:    nslink++;        break;
133                 case S_IFSOCK:   nsock++;         break;
134                 case S_IFDIR:
135                         printf("for S_IFDIR for %s\n", pathname);
136                 }
137                 break;
138 
139         case FTW_D:
140                 ndir++;
141                 break;
142         case FTW_DNR:
143                 printf("can't read directory %s\n", pathname);
144                 break;
145         case FTW_NS:
146                 printf("stat error for %s\n", pathname);
147                 break;
148         default:
149                 printf("unknown type %d for pathname %s\n", type, pathname);
150         }
151         return(0);
152 }
```

在终端中使用 gcc 对其进行编译并且运行，分别对当前工作目录以及工作目录的上一层目录进行统计，可以看到如下的输出。

```
alloy@ubuntu:~/linuxc/chapter4$ gcc exam404readdir.c -o exam404readdir
alloy@ubuntu:~/linuxc/chapter4$ ./exam404readdir /home/alloy/linuxc/chapter4
普通文件 =          8, 80.00 %
目录文件 =          2, 20.00 %
块设备文件 =          0,   0.00 %
字设备文件 =          0,   0.00 %
FIFOs =          0,   0.00 %
符号链接文件 =          0,   0.00 %
套接字文件 =          0,   0.00 %
alloy@ubuntu:~/linuxc/chapter4$ ./exam404readdir /home/alloy/linuxc
普通文件 =         26, 81.25 %
目录文件 =          6, 18.75 %
块设备文件 =          0,   0.00 %
字设备文件 =          0,   0.00 %
FIFOs =          0,   0.00 %
符号链接文件 =          0,   0.00 %
套接字文件 =          0,   0.00 %
```

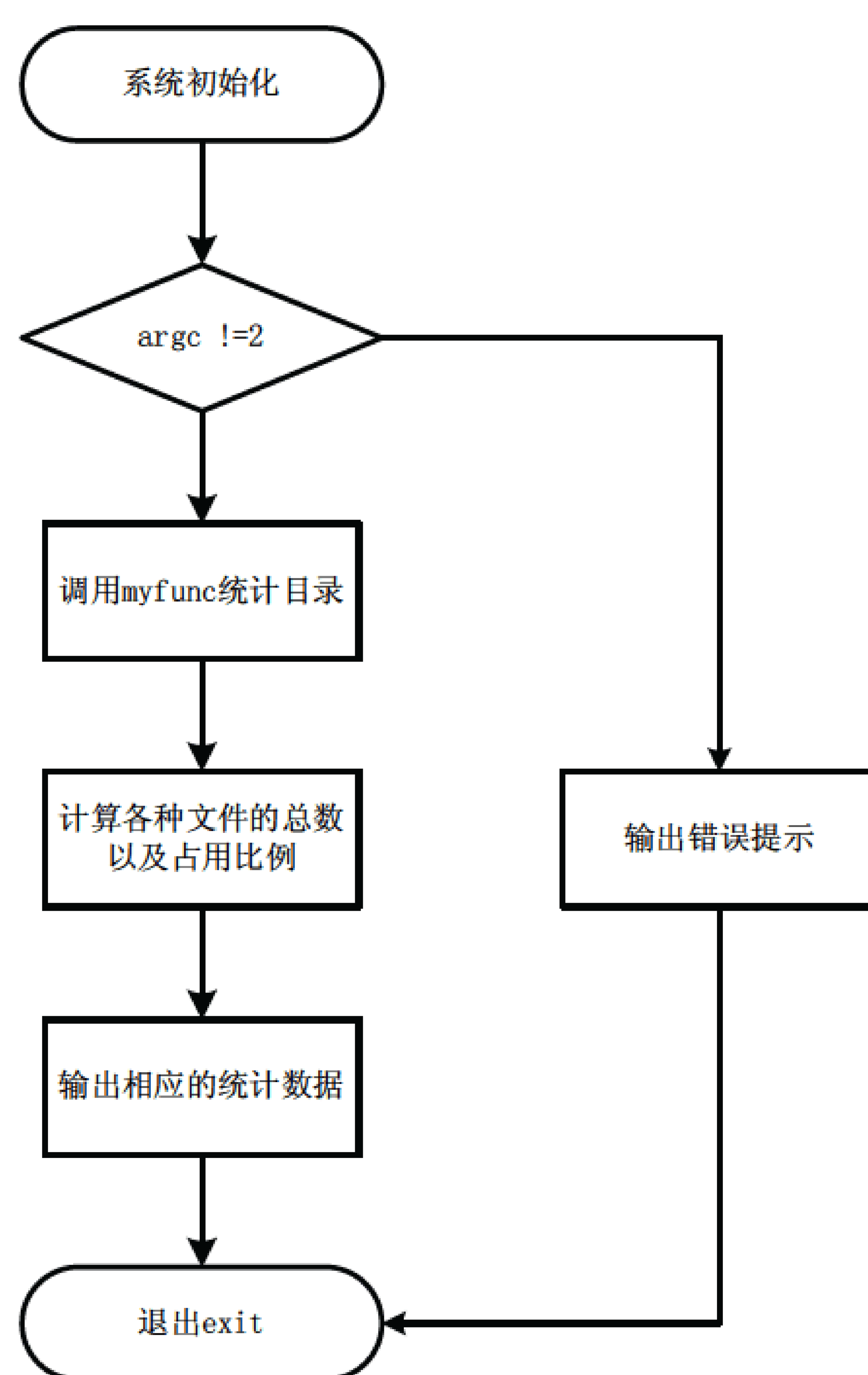


图 4.7 统计目录文件

4.1.3 当前工作目录路径

在 Linux 中，每个进程都有一个对应的工作目录，也就是常说的当前工作路径，例如如下是一个当前工作路径的实例，可以使用 pwd 命令在 Shell 下获取这个工作路径：

```
/home/alloy/linuxc/chapter4
```

进程可以调用 chdir 函数或者 fchdir 函数来修改当前的工作目录，对 chdir 函数和 fchdir 函数的标准调用格式说明如下，如果调用成功则返回 0，如果出错则返回-1。

```
#include <unistd.h>
int chdir (const char *patnname);
int fchdir (int fd);
```

在 chdir 函数和 fchdir 函数中，分别使用目录的路径以及目录的文件描述符来作为参数。


注 意

调用 chdir 函数的进程必须具有 pathname 所有路径分量的执行权限，并且 pathname 指定的路径长度不能超过 PATH_MAX，其路径分量不能超过 NAME_MAX。

在某些应用中，用户需要获取当前工作目录的完整路径（绝对路径），此时可以使用 getcwd 函数，对其标准调用格式说明如下，如果成功调用则返回当前的目录路径，如果失败则返回 NULL 空指针。

```
#include <unistd.h>
char *getcwd (char *buf, size_t size);
```

getcwd 函数的参数 buf 是用于存放路径的缓冲地址，而 size 存放的是缓冲长度（单位是字节）。

getcwd 函数从当前工作目录（.目录项）开始，利用“..”目录项找到其上一级的目录，然后读其目录项，直到该目录项中的 i 节点编号数与工作目录 i 节点编号数相同，这样就找到了其对应的文件名。按照这种方法逐层上移，直到遇到根，这样就得到了当前工作目录的绝对路径名。

【例 4.6】切换当前工作目录路径

例 4.6 是 chdir 函数和 getcmd 函数的应用实例，应用代码调用 mkdir 函数在指定的文件夹下建立一个新的文件夹，然后使用 chdir 切换工作目录到该文件夹下，打印输出当前的工作目录，然后在该工作目录下建立一个新的目录，其工作流程如图 4.8 所示。

实例的应用代码如下：

```
1    //这是一个 chdir 和 getcwd 函数的应用实例
2    //首先使用 mkdir 函数在当前文件夹下建立一个新的文件夹
3    //然后使用 chdir 函数切换工作目录到新建的文件夹下
4    //打印输出切换后的工作路径，然后在该工作目录下建立一个新的文件夹
5    #include <stdio.h>
6    #include <unistd.h>
7    #include <fcntl.h>
8    int main(int argc,char *argv[])
9    {
10       unsigned char temp;
11       char npath[200];                          //路径字符串缓冲区
12       if(argc != 3 )                            //如果参数不为 3
13       {
14           perror("请输入正确的参数!\n");        //参数错误
15           return 1;                             //退出
16       }
17       temp = mkdir(*(argv+1),S_IRUSR|S_IWUSR|S_IXUSR);
18       //在当前工作路径（文件夹下）下新建一个目录，目录名由 arvg[1]指定
19       if(temp == -1)          //如果创建失败
20       {
21         printf("创建文件失败！ /n");             //创建目录失败
22         return 2;                               //退出
23       }
24       temp = chdir(*(argv+1));                  //切换目录到 arvg[1]指定的目录下
25       if(temp == -1)                            //切换目录失败
26       {
27         printf("切换目录操作失败！ /n");
28         return 3;
29       }
30       else                                          //切换目录操作成功
31       {
32         if(getcwd(npath,200) == NULL)               //如果没有获得当前的工作路径
33         {
34           printf("不能获得当前的工作路径！ \n");
35           return 4;
36         }
37         else
38         {
39           printf("当前的工作路径是 %s\n",npath);    //打印输出当前的工作路径
40         }
41       }
42       temp = mkdir(*(argv+2),S_IRWXU|S_IRGRP|S_IXOTH);
43       //再建立一个由 argv[2]指定名称的文件夹
```

```
44        return 0;
45    }
```

在终端中使用 gcc 对其进行编译并且运行，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter4$ gcc exam405chdirgetcmd.c -o exam405chdirgetcmd
//编译链接生成可执行文件
alloy@ubuntu:~/linuxc/chapter4$ ./exam405chdirgetcmd dir1 dir2
//在当前工作目录下创建目录 dir1，然后将当前工作目录路径切换到 dir1，再创建
//目录 dir2，最后输出当前工作目录路径
当前的工作路径是 /home/alloy/linuxc/chapter4/dir1
alloy@ubuntu:~/linuxc/chapter4$ ls
//使用 ls 命令查看新建立的目录，可以看到 dir1
dir1              exam402rmdir     exam403opendir.c   exam405chdirgetcmd
exam401mkdir      exam402rmdir.c   exam404readdir     exam405chdirgetcmd.c
exam401mkdir.c    exam403opendir   exam404readdir.c   opendirtest
```

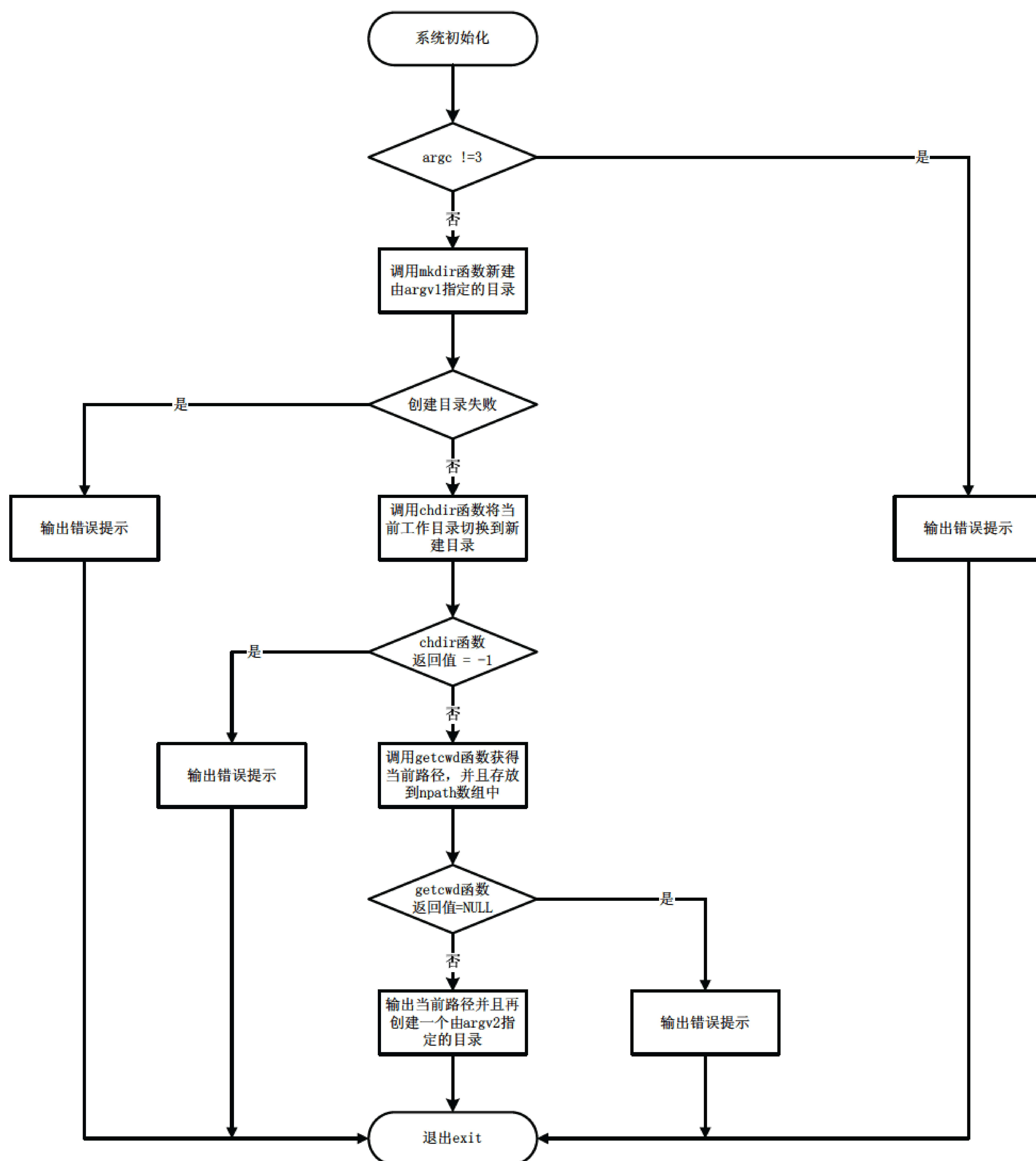


图 4.8 切换当前工作目录路径

4.2 目录文件的综合应用——定时创建目录和文件

在上一节中介绍了目录文件的基础操作方法，本节将介绍一个在 Linux 下进行目录操作的综合应用实例。

4.2.1 综合应用的需求分析

本应用使用当前时间的“时+分”信息为名称参数创建一个目录文件，然后在该目录文件中以“秒”信息为名称参数创建一个文件，在实际应用中可以用于以时间为参数保存“大流量数据”，例如某个数据源每秒钟都产生大量的数据，需要以秒为单位将这些数据保存到 Linux 中，并且希望将每分钟的数据都放到同一个文件夹下，应用的工作流程如图 4.9 所示，在实现过程中需要考虑如下方面的内容。

- 以秒为单位获取当前的时间信息。
- 使用“时+分”信息作为名称参数来创建目录文件，使用“秒”信息作为名称参数来创建普通文件。
- 当前工作目录路径的切换。
- 各个模块的综合。

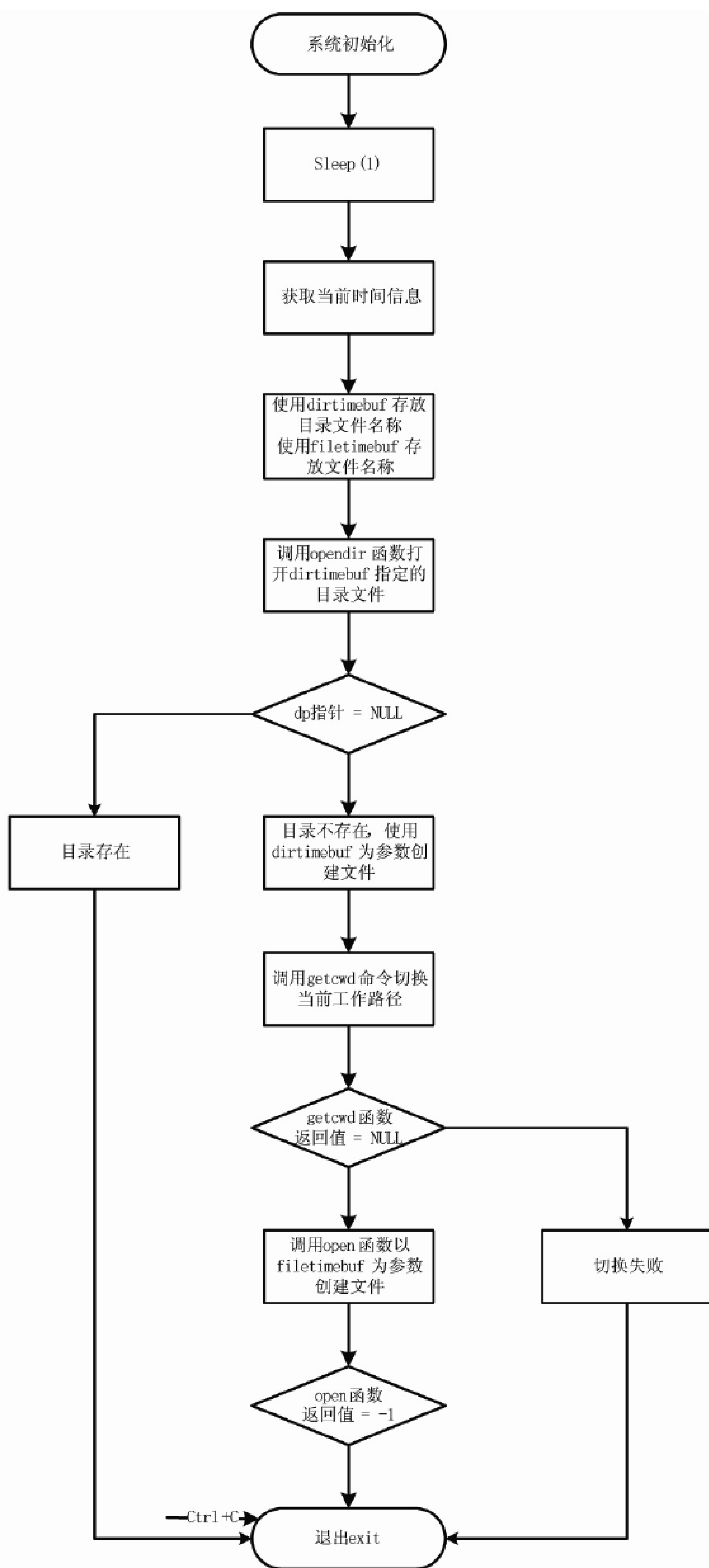


图 4.9 定时创建目录和文件流程

4.2.2 使用时间信息生成目录和文件

这是完成需求的第一步，目的是使用当前的时间信息分别建立对应的目录以及目录下的文件，其应用代码如例 4.7 所示，目录和文件的名称信息字符串分别存放在 filenamebuf 和 dirnamebuf 数组中，生成该字符串的方法可以参考第 3.3.4 小节的例 3.17。

【例 4.7】使用时间信息生成目录和文件

实例的应用代码如下：

```
1   //使用当前时间的“时+分”信息为名称来创建一个文件夹
2   //然后在该文件夹下以“秒”信息为名称来创建一个文件
3   //需要判断文件夹和文件是否存在
4   #include <time.h>
```

```
5   #include <stdio.h>
6   #include <string.h>
7   #include <fcntl.h>
8   #include <sys/stat.h>
9   #include <sys/types.h>
10  #include <dirent.h>
11  #include <unistd.h>
12
13  int main(int argc,char *argv)
14  {
15     time_t timetemp;                    //定义一个时间结构体变量
16     struct tm *p;                       //结构体指针
17     DIR *dp;                            //目录文件指针
18     int temp = 0;                       //存放 mkdir 函数的返回值
19     int chdirtemp = 0;                  //存放 chdir 函数的返回值
20     int fd;                             //文件描述符
21     char filetimebuf[3];                //目录时间信息
22     char dirtimebuf[5];                 //文件时间信息
23     char dirnamebuf[10] = "DIR";        //目录名缓冲区
24     char filenamebuf[10] = "File";      //文件名缓冲区
25     char npath[100];                    //当前工作目录的完整路径
26     time(&timetemp);                    //获得时间参数
27     printf("当前时间为%s",asctime(gmtime(&timetemp)));
28     p = localtime(&timetemp);
29     printf("小时 =%d ,分 =%d ,秒 =%d\n",p->tm_hour,p->tm_min,p->tm_sec);
       //输出一次当前的时、分、秒信息
30     sprintf(dirtimebuf,"%02d%02d",p->tm_hour,p->tm_min);
31     sprintf(filetimebuf,"%02d",p->tm_sec);
32     //将时、分、秒信息按照 2 位前端补 0 的方式格式化送入目录和文件时间 buf
33     strcat(filenamebuf,filetimebuf);
34     strcat(dirnamebuf,dirtimebuf);
35     //生成文件和目录名称，存放到对应的缓冲区中
36     printf("%s\n",filenamebuf);
37     printf("%s\n",dirnamebuf);
38     dp = opendir(dirnamebuf);           //尝试打开目录
39     if(dp == NULL)                      //出错，说明目录不存在
40     {
41        printf("目录%s 不存在\n",dirnamebuf);
42        temp = mkdir(dirnamebuf,S_IRWXU|S_IRGRP|S_IXOTH);     //尝试创建目录
43        if(temp == -1)                   //创建目录失败
44        {
45           printf("创建目录失败。\n");
46           return 1;
47        }
48        else                             //创建目录成功
49        {
50           printf("创建目录%s 成功\n",dirnamebuf);
51           chdirtemp = chdir(dirnamebuf);              //将当前工作目录切换到新建的目录下
52           if(chdirtemp == -1)                         //表明切换失败
```

```
53              {
54                printf("切换当前工作目录失败\n");
55                return 2;
56              }
57              else                              //切换当前工作目录成功，创建文件
58              {
59                if(getcwd(npath,100) == NULL)   //如果已经获得当前的工作目录则打印输出，否则退出
60                {
61                  printf("未能获得当前工作目录路径\n");
62                  return 3;
63                }
64                else
65                {
66                  printf("当前工作目录的完整路径是%s\n",npath);
67                }
68                fd = open(filenamebuf,O_RDWR|O_CREATE,S_IRWXU);        //创建文件
69                if(fd != -1)                                           //表明创建文件成功
70                {
71                  printf("创建文件%s 成功\n",filenamebuf);
72                  close(fd);                                           //关闭文件
73                }
74                else
75                {
76                  printf("创建文件失败\n");
77                  return 4;
78                }
79              }
80          }
81      }
82      else                                              //能打开目录，则表明目录存在
83      {
84          printf("目录%s 已经存在\n",dirnamebuf);
85          closedir(dp);                                 //关闭目录
86          //接下来切换当前工作目录到已经存在的目录，创建文件
87          chdirtemp = chdir(dirnamebuf);                //将当前工作目录切换到新建的目录下
88          if(chdirtemp == -1)                           //表明切换失败
89          {
90              printf("切换当前工作目录失败\n");
91              return 2;
92          }
93          else                                          //切换当前工作目录成功，创建文件
94          {
95              if(getcwd(npath,100) == NULL)  //如果已经获得当前的工作目录，则打印输出，否则退出
96              {
97                printf("未能获得当前工作目录路径\n");
98                return 3;
99              }
100             else
101             {
```

```
102                printf("当前工作目录的完整路径是%s\n",npath);
103            }
104            fd = open(filenamebuf,O_RDWR|O_CREATE,S_IRWXU);        //创建文件
105            if(fd != -1)                                          //表明创建文件成功
106            {
107                printf("创建文件%s 成功\n",filenamebuf);
108                close(fd);                                        //关闭文件
109            }
110            else
111            {
112                printf("创建文件失败\n");
113                return 4;
114            }
115        }
116    }
117    return 0;
118 }
```

在终端中使用 gcc 对其进行编译并且运行，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter4$ gcc exam406timeopendir.c -o exam406timeopendir
//编译链接，生成可执行文件
alloy@ubuntu:~/linuxc/chapter4$ ./exam406timeopendir
当前时间为 Tue Feb 18 14:18:48 2014
小时 ＝22 ,分 ＝18 ,秒 ＝48
File48
DIR2218
目录 DIR2218 不存在
创建目录 DIR2218 成功
当前工作目录的完整路径是/home/alloy/linuxc/chapter4/DIR2218
创建文件 File48 成功
alloy@ubuntu:~/linuxc/chapter4$ ls
//此时已经在当前工作目录下创建了目录文件 DIR2218，并且在该目录下创建了文件
//FILE48
dir1            exam401mkdir.c  exam403opendir     exam404readdir.c       exam406timeopendir
DIR2218         exam402rmdir    exam403opendir.c   exam405chdirgetcmd     exam406timeopendir.c
exam401mkdir    exam402rmdir.c  exam404readdir     exam405chdirgetcmd.c   opendirtest
```

4.2.3　定时创建目录和文件

在例 4.8 的基础上参考第 3.3.3 小节中的例 3.16 增加对应的定时功能即可，其应用代码如例 4.8 所示。

【例 4.8】定时创建目录和文件

实例的应用代码如下：

```
//使用当前时间的“时+分”信息为名称创建一个文件夹
//然后在该文件夹下以“秒”信息为名称创建一个文件
//需要判断文件夹和文件是否存在
```

```
#include <time.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>
#include <unistd.h>
int main(int argc,char *argv)
{
  while(1)
  {
  time_t timetemp;              //定义一个时间结构体变量
  struct tm *p;                 //结构体指针
  DIR *dp;                      //目录文件指针
  int temp = 0;                 //存放 mkdir 函数的返回值
  int chdirtemp = 0;            //存放 chdir 函数的返回值
  int fd;                       //文件描述符
  char filetimebuf[3];          //目录时间信息
  char dirtimebuf[5];           //文件时间信息
  char dirnamebuf[10] = "DIR";  //目录名缓冲区
  char filenamebuf[10] = "File"; //文件名缓冲区
  char npath[100];              //当前工作目录的完整路径
    sleep(1);                   //简单的使用秒延时，每隔 1 秒执行一次
    time(&timetemp);            //获得时间参数
    printf("当前时间为%s",asctime(gmtime(&timetemp)));
    p = localtime(&timetemp);
    printf("小时 = %d ,分 = %d ,秒 = %d\n",p->tm_hour,p->tm_min,p->tm_sec);
    //输出一次当前的时、分、秒信息
    sprintf(dirtimebuf,"%02d%02d",p->tm_hour,p->tm_min);
    sprintf(filetimebuf,"%02d",p->tm_sec);
    //将时、分、秒信息按照 2 位前端补 0 的方式格式化送入目录和文件时间 buf
    strcat(filenamebuf,filetimebuf);
    strcat(dirnamebuf,dirtimebuf);
    //生成文件和目录名称，存放到对应的缓冲区中
    printf("%s\n",filenamebuf);
    printf("%s\n",dirnamebuf);
    dp = opendir(dirnamebuf);         //尝试打开目录
    if(dp == NULL)                    //出错，说明目录不存在
    {
      printf("目录%s 不存在\n",dirnamebuf);
      temp = mkdir(dirnamebuf,S_IRWXU|S_IRGRP|S_IXOTH);       //尝试创建目录
      if(temp == -1)                  //创建目录失败
      {
        printf("创建目录失败。\n");
        return 1;
      }
      else                            //创建目录成功
     {
```

```
        printf("创建目录%s 成功\n",dirnamebuf);
        chdirtemp = chdir(dirnamebuf);          //将当前工作目录，切换到新建的目录下
        if(chdirtemp == -1)                     //表明切换失败
        {
          printf("切换当前工作目录失败\n");
          return 2;
        }
        else                                    //切换当前工作目录成功，创建文件
        {
          if(getcwd(npath,100) == NULL)  //如果已经获得当前的工作目录，则打印输出，否则退出
          {
            printf("未能获得当前工作目录路径\n");
            return 3;
          }
          else
          {
            printf("当前工作目录的完整路径是%s\n",npath);
          }
          fd = open(filenamebuf,O_RDWR|O_CREATE,S_IRWXU);    //创建文件
          if(fd != -1)                          //表明创建文件成功
          {
            printf("创建文件%s 成功\n",filenamebuf);
            close(fd);                          //关闭文件
          }
          else
         {
            printf("创建文件失败\n");
            return 4;
         }
        }
      }
    }
    else                                        //若能打开目录，则表明目录存在
    {
      printf("目录%s 已经存在\n",dirnamebuf);
      closedir(dp);                             //关闭目录
      //接下来切换当前工作目录到已经存在的目录，创建文件
      chdirtemp = chdir(dirnamebuf);            //将当前工作目录切换到新建的目录下
      if(chdirtemp == -1)                       //表明切换失败
      {
        printf("切换当前工作目录失败\n");
        return 2;
      }
      else                                      //切换当前工作目录成功，创建文件
      {
        if(getcwd(npath,100) == NULL)  //如果已经获得当前的工作目录，则打印输出，否则退出
        {
          printf("未能获得当前工作目录路径\n");
          return 3;
```

```
                }
                else
                {
                    printf("当前工作目录的完整路径是%s\n",npath);
                }
                fd = open(filenamebuf,O_RDWR|O_CREATE,S_IRWXU);        //创建文件
                if(fd != -1)                                            //表明创建文件成功
                {
                    printf("创建文件%s 成功\n",filenamebuf);
                    close(fd);                                          //关闭文件
                }
                else
                {
                    printf("创建文件失败\n");
                    return 4;
                }
            }
        }
    }                                                                   //循环结束
        return 0;
}
```

4.3 本章习题

1. 编写一个程序，将当前工作目录修改为用户指定的目录，然后调用 getcwd 命令获取并且打印当前的目录路径，

2. 编写一个应用程序，首先用 getcwd 函数取得当前工作目录，然后在当前工作目录下，利用 mkdir 函数创建新目录。新目录创建成功后，改变当前工作目录为新目录，然后切换回上一级目录后删除新创建的目录。

3. 编写一个程序，首先用 opendir 函数打开用户指定的目录，然后调用 readdir 函数读取该目录内容，并打印出所读取的目录内容，最后用 closedir 函数将刚才打开的目录文件关闭。

4. 编写一个程序，在某指定路径下创建一个空目录 temp，该目录文件的访问权限为用户可读可写可执行，同组用户可读可执行，其他组用户可读可执行，并返回函数执行的结果。

第 5 章　Linux 的文件系统和文件属性

文件系统是 Linux 系统在其磁盘空间上组织和管理文件的方法和数据结构，其由与文件管理有关的软件、被管理文件以及实施文件管理所需的数据结构三个部分组成，负责为用户建立文件，存入、读出、修改、转储文件，控制文件的存取，当用户不再使用时撤销文件等，文件系统下的每个文件都有其对应的属性。在第 3 章中介绍了 Linux 文件的基础分类，第 4 章介绍了目录文件的操作方法，本章将介绍其他一些特殊文件的操作方法，涉及的内容包括：

- Linux 的文件系统和文件属性基础。
- 在 Linux 中使用 C 语言对文件属性进行操作的方法，这些属性包括文件的类型、文件的时间、文件的权限、文件的名称等。
- 在 Linux 中使用 C 语言删除文件的方法。
- 在 Linux 中使用对链接文件进行操作的方法。

5.1　Linux 的文件系统和文件属性基础

从系统角度来看，文件系统是对文件存储器空间进行组织和分配，负责文件存储并对存入的文件进行保护和检索的系统。其负责为用户建立文件，存入、读出、修改、转储文件，控制文件的存取，当用户不再使用时撤销文件等操作。

不同的操作系统，其支持的文件系统和文件格式不同，可能存在某些文件拿到另外一个系统中就打不开、看不到等情况，如图 5.1 所示。

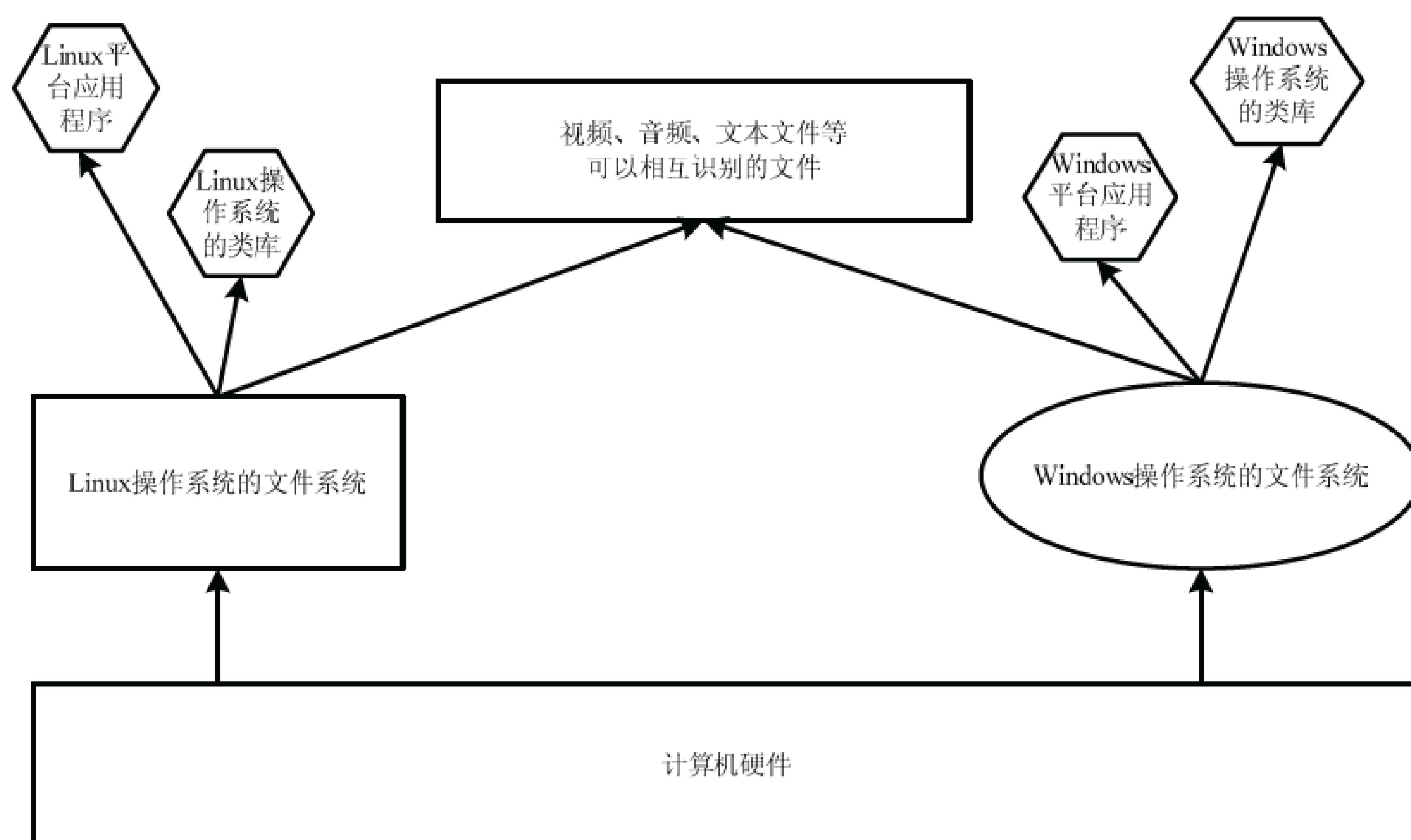


图 5.1　操作系统和文件

可以使用 df 命令来查看当前 Linux 系统的文件系统信息，其执行过程如下，在其中可以看到相应的容量、已用、可用和挂载点等信息。

```
alloy@ubuntu:~$ df -h
文件系统          容量    已用    可用    已用%   挂载点
/dev/loop0        29G     5.3G    23G     19%     /
udev              1.9G    4.0K    1.9G    1%      /dev
tmpfs             751M    996K    750M    1%      /run
none              5.0M    0       5.0M    0%      /run/lock
none              1.9G    76K     1.9G    1%      /run/shm
/dev/sda3         98G     9.1G    89G     10%     /host
```

5.1.1 Linux 的文件系统

在第 1.4 节中介绍过 Linux 系统支持多种文件系统，包括 ext2、ext3、vfat、ntfs、iso9660、Jffs、Yaffs/Yaffs2、Romfs 和 NFS 等，本小节将介绍一些常用的文件系统。

1. ext 文件系统

于 1992 年 4 月发布的 ext（扩展文件系统）是第一个专门为 Linux 设计的文件系统，其为 Linux 的发展做出了重要的贡献，但是在性能和兼容性上存在许多缺陷，现在已经基本不使用了。

2. ext2 文件系统

于 1993 年发布的 ext2（二级扩展文件系统）是为解决 ext 文件系统的缺陷而设计的可扩展的高性能文件系统。其曾经是 Linux 系统上应用最为广泛的文件系统，在 2000 年以前几乎所有的 Linux 发行版都用 ext2 作为默认的文件系统，是 GNU/Linux 系统中标准的文件系统，其特点为存取文件（尤其是中小型文件）的性能很好，在速度和处理器利用率的优势上较为突出。

ext2 支持 256 字节的长文件名，其单一文件大小与文件系统本身的容量上限、文件系统本身的簇大小有关，例如在 x86 兼容处理器的系统中，簇最大为 4KB，则单一文件大小上限为 2048GB，而文件系统的容量上限为 6384GB。

ext2 也有其自身的缺点，它的设计者主要考虑的是文件系统性能方面的问题，其在写入文件内容的同时并没有写入文件的 meta-data（和文件有关的信息，例如权限、所有者以及创建和访问时间），也就是说 Linux 系统先写入文件的内容，然后等到有空的时候才写入文件的这些数据，如果在这个时间间隙中出现了意外情况（例如系统断电），此时就会造成文件系统处于不一致的状态。

3. ext3 文件系统

ext3 文件系统是 ext2 的升级版本，从 ext2 向 ext3 迁移非常方便，被称为 ext2 的“下一个版本”，其在 ext2 文件系统的基础上加入了记录元数据的日志功能，努力保持向前和向后的兼容性。

ext3 文件系统是一种日志式文件系统，其优越性在于由于文件系统都有快取层参与运行，如不使用时必须将文件系统卸下以便将快取层的数据写回磁盘中，因此每当系统要关机时就必须将其所有的文件系统全部卸下后才能进行关机，如果在文件系统尚未卸下前就关机（如忽然掉电时），则会造成文件系统的资料不一致，则必须做文件系统的重整工作，即将不一致与错误的地方修复。

ext3 文件系统的最大缺点是没有现代文件系统所具有的能提高文件数据处理速度和解压的高

性能，另外使用 ext3 文件系统时要注意硬盘限额问题。



注 意

本书介绍的 Ubuntu 12.04 lts 使用的就是 ext3 文件系统。

4. JSF 文件系统

JSF（Journaled File System Technology for Linux）文件系统是基于日志的字节级文件系统，其是为面向事务的高性能系统而开发的。2000 年 2 月，IBM 宣布在一个开放资源许可证下移植 Linux 版的 JSF 文件系统。

JSF 主要是为满足服务器（从单处理器系统到高级多处理器和群集系统）的高吞吐量和可靠性需求而设计的，其具有可伸缩性和健壮性，与非日志文件系统相比，它的优点是其快速重启能力。由于使用了数据库日志处理技术，JSF 能在几秒或几分钟之内把文件系统恢复到一致状态，而在非日志文件系统中，这个动作可能需要花费几小时或几天的时间。

JSF 的缺点在于由于需要保持一个日志数据，所以系统需要写入许多数据，占用的系统资源比较高，对系统的硬件系统会造成一定的损失。

5. 其他文件系统

其他常见的文件系统还包括 Minix、FAT 系列、NTFS 等，对它们的简要说明如下。

- Minix：Linux 支持的第一个文件系统，对用户有很多限制而且性能低下，例如文件没有时间标记、支持文件名最长为 14 个字符等；其最大的缺点是最大只能使用 64MB 的硬盘分区。
- Xia: Minix 文件系统修正后的版本，在一定程度上解决了文件名和文件系统大小的局限。
- msdos：这是在 DOS、Windows 和某些 OS/2 操作系统上使用的一种文件系统，其文件名称采用“8+3”的形式，即 8 个字符的文件名加上 3 个字符的扩展名。
- umsdos：这是在 Linux 下扩展 msdos 文件系统的驱动，支持长文件名、所有者、允许权限、连接和设备文件，允许一个普通的 msdos 文件系统用于 Linux，而且无须为它建立单独的分区。
- iso9660：标准 CDROM 文件系统，通用的 Rock Ridge 增强系统，允许长文件名。
- vfat：这是 Windows 操作系统下使用的一种文件系统，其在 msdos 文件系统的基础上增加了对长文件名的支持。
- nfs：Sun 公司推出的网络文件系统，允许多台计算机之间共享同一文件系统，易于从所有这些计算机上存取文件。
- hpfs：高性能文件系统（High Performance File System）是微软的 LAN Manager 中的文件系统，同时也是 IBM 的 LAN Server 和 OS/2 的文件系统，其能访问较大的硬盘驱动器，提供更多的组织特性并改善了文件系统的安全特性。
- smb: smb 是一种支持 Windows for workgroups、Windows NT 和 Lan Manager 的基于 SMB 协议的网络操作系统。

- Sysv：该文件系统实际上是 System V/Coherent 在 Linux 平台上的文件系统。
- Ncpfs：这是一种 Novell NetWare 使用的 NCP 协议的网络操作系统。
- Proc：这是 Linux 系统中作为一种伪文件系统出现的文件系统，通常用来作为连接内核数据结构的界面。
- FAT 系列文件系统：包括 FAT12、FAT16、FAT32 等文件系统，通常应用在微软公司的 Windows 操作系统下。
- NTFS：这是微软 Windows NT 内核的系列操作系统支持的一个特别为网络和磁盘配额、文件加密等管理安全特性设计的磁盘格式。

5.1.2 Linux 的文件系统结构

和 DOS 或者 Windows 操作系统类似，Linux 采用了目录树的格式来管理所有的文件和目录（在 Linux 中目录其实也是一个文件，这点将在后续中介绍）；和 DOS 或者 Windows 不同的是 Linux 的树形目录中只有唯一的一个根目录"/"（称为根，root），其他目录都是这个根目录衍生的子目录，图 5.2 和图 5.3 分别是 DOS/Windows 和 Linux 的目录树文件结构。

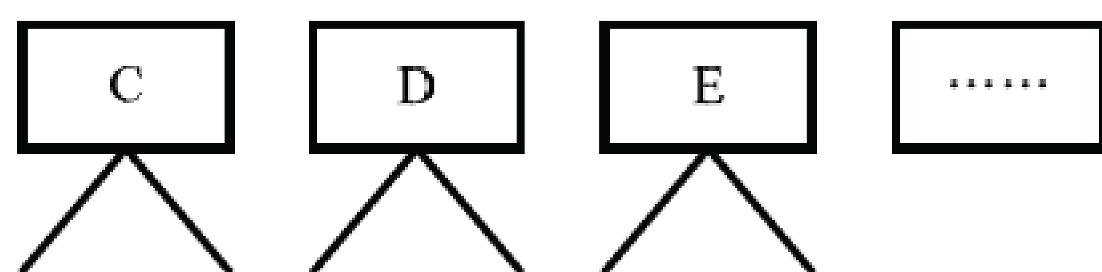


图 5.2 DOS/Windows 的目录树文件结构

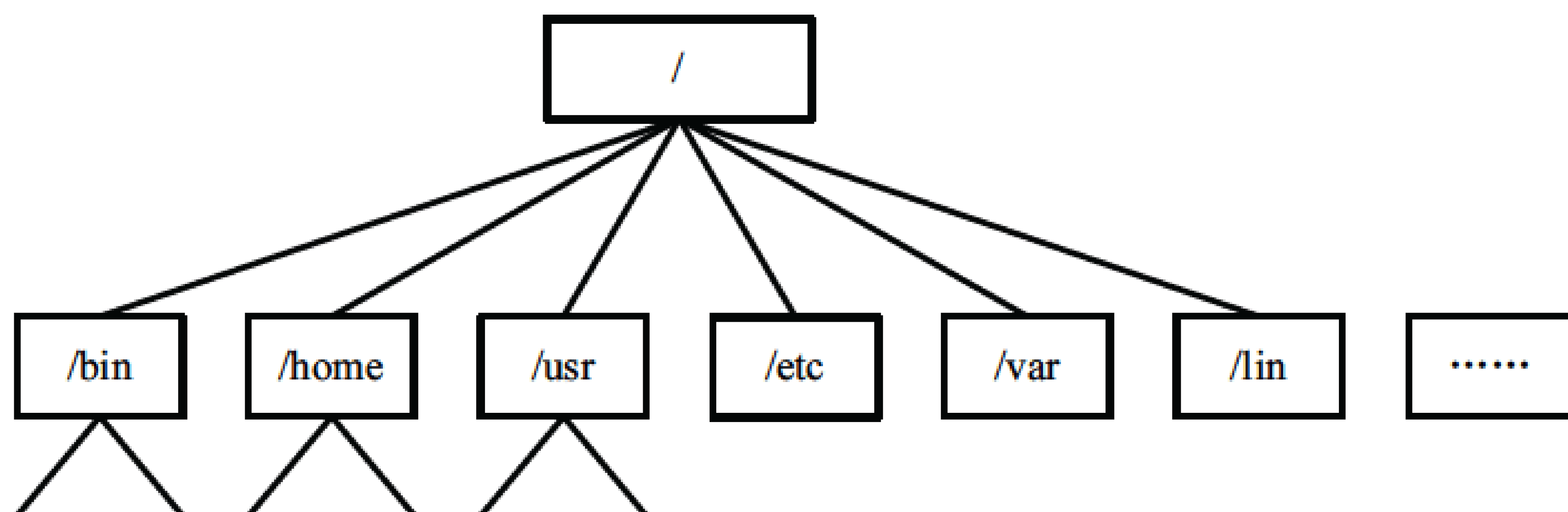


图 5.3 Linux 的目录树文件结构



注 意

在 Linux 中，所有的内容都被看成文件，包括硬件和目录；所有的操作都可以归结为对文件的操作，Linux 可以像操作普通文件一样来对磁盘文件、串口、键盘、显示器、打印机以及其他的设备进行操作。

Linux 的文件系统是目录和文件的一种层次安排，目录的起点称为根（root），即一个字符"/"；目录（directory）是一个包含目录项的文件，在逻辑上，可以认为每个目录项都包含一个文件名，同时还包含说明该文件属性的信息。通过这种树形等级结构，用户可以浏览整个系统，可以进入任何一个已授权进入的目录并且访问相应的文件，以下是对根目录下一些目录的说明。

- bin：存放系统启动时需要的执行文件以及一些用户常用的命令，例如 cp、ls、cat 等。
- boot：存放系统内核以及启动管理器，类似于 grub。
- cdrom：Ubuntu 系统安装光盘镜像的挂载位置，这个目录根据用户的实际情况可能不会

存在。

- dev：设备文件目录，在其中存放了相应的设备信息。
- etc：存放相应的系统配置文件。
- home：用户主目录，在其中按照用户名存放了当前系统中存在用户的个人文件和信息，类似于 Windows 下的“我的文档”。
- lib：存放着系统最基本的动态链接共享库，其作用类似于 Windows 里的.dll 文件。
- lib64：这是 lib 目录的 64 位版本，当使用 64 位的操作系统时会存在这个目录，并且将对应的 64 位库函数存放于其中。
- lost+found：存放文件系统修复时恢复的文件。
- media：用于存放 Ubuntu 系统加载的各种媒体，例如光盘、软盘等，在其他 Linux 操作系统中可能不存在。
- mnt：用户临时挂载其他的文件系统，如挂接 U 盘、CD-ROM 等。
- opt：用于存放安装时“可选”的程序，例如各种图形界面 KDE、Gnome 等。
- proc：系统内存的映射虚拟目录，可以通过直接访问这个目录来获取系统信息，其是存在于内存中而不是硬盘上的。
- root：root 用户的主工作目录，类似于 home。
- run：存放的是自系统启动以来描述系统信息的文件，在某些 Linux 中这个目录可能位于 var 下。
- sbin：存放系统级的可执行文件，类似于 bin，但是这些文件只能让 root 用户而不能让普通用户使用。
- selinux：存放提供强制访问控制的相应文件，在某些 Linux 中可能不存在。
- srv：存放提供一些特定服务的文件。
- sys：存放系统信息的相关文件。
- tmp：存放临时文件。
- usr：存放普通用户的应用程序、文档、程序等。
- var：存放在时间、大小、内容上会经常变化的文件。


注 意

和 Windows 用户自己建立相应的文件夹来对所有的文件进行分门别类的管理不同，Linux 提供了相应的文件夹来主动对文件进行分类管理。

Linux 的文件系统由如下 4 部分组成：引导块、超级块、索引节点表和数据块，对各个部分的详细说明如下。

- 引导块：用于存放文件系统的引导程序，引导程序是用于系统引导或启动操作系统。如果一个文件系统不存放操作系统，其引导块将为空。
- 超级块：用来描述该文件系统管理的资源，其包含空闲索引节点表和空闲数据块表，用于具体说明文件系统的资源使用情况。
- 索引节点表：用来存储文件的控制信息，每个节点对应一个文件。

- 数据块：是磁盘上存放数据的磁盘块，包括目录文件和数据。

图 5.4 是 Linux 文件系统的组织结构。

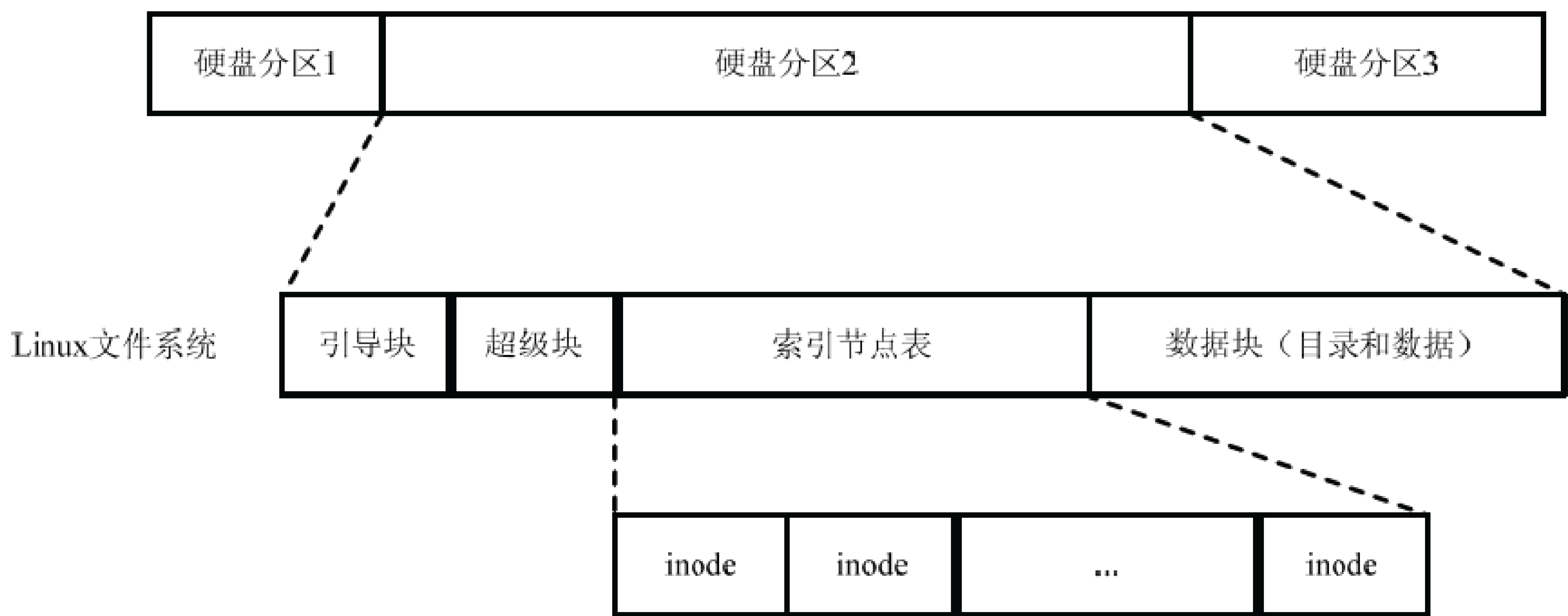


图 5.4　Linux 文件系统组织结构

这 4 个部分中最重要的是超级块和索引节点表，它们都是用于描述当前文件系统状态的组成。

超级块用于描述 Linux 文件系统的资源状态，包括文件系统的大小、空闲单元位置信息等，在文件系统对文件的管理中起到至关重要的作用，其由如下字段构成：

- 文件系统的容量信息，如数据块数目、保留块数目和块的大小等。
- 文件系统中空闲块的数目。
- 文件系统中部分可用的空闲块表。
- 空闲块表中下一个空闲块号。
- 索引节点表的大小。
- 文件系统中空闲索引节点表数目。
- 文件系统中部分空闲索引节点表。
- 空闲索引节点表中下一个空闲索引节点号。
- 超级块的锁字段，用于保证对存储单元的互斥操作。
- 空闲块表的锁字段和空闲索引节点的锁字段。
- 超级块是否被修改的标志。
- 其他字段，存放了文件系统是否完整的标志。



注　意

在 Linux 关机的时候要求先将缓冲区数据写回文件系统，并且卸载该文件系统，如果没有卸载文件系统就关机，则很可能导致数据丢失；而在 Linux 启动的时候，在挂载一个文件系统之前首先会去检查其超级块中的相应字段，如果上次没有进行卸载操作，则需要对该文件系统的完整性进行检查。

超级块给出的是文件系统的相关信息，而一个文件信息则是由索引节点表（inode）给出，每个文件都有自己的索引节点表，在其中包含了该文件数据在磁盘上存储的位置、操作权限、文件所有者、操作时间等信息。

索引节点表平时存储在磁盘上，在需要进行操作的时候读入内存，通常来说存储在磁盘上的索引节点表称作磁盘索引节点，而把其在内存中的映像称作内存索引节点表。

索引节点表由如下字段构成。

- 文件类型：Linux 的文件可以分为普通文件、目录文件、链接文件、设备文件、管道文件等，将在下一个小节进行详细介绍。
- 文件链接数：记录了引用该文件的目录表项数，即记录了有多少个文件名指向该文件。
- 文件属主标识：指出该文件的所有者 id。
- 文件属主的组标识：指出该文件所有者属组的 id。
- 文件的访问权限：系统将用户分为文件属主、同组用户和其他用户三类，每类用户可能获得对文件中的一种或几种访问权限。需要特别指出的是，目录文件的执行权限是指修改目录的权力。
- 文件的存取时间：包括文件最后一次被修改的时间、最后一次被访问的时间和最后一次修改索引节点的时间。
- 文件的长度：以字节表示的文件长度。
- 文件的数据块指针：文件操作的当前位置指针。

在索引节点表中并不包含文件的名称，文件名的信息是存放在目录文件中，其具体存放方式将在下一小节中介绍。

在 Linux 中的 stat.h 头文件中使用了一个结构体来定义索引节点表的相应字段，对其说明如下：

```
#ifndef _ALPHA_STAT_H
#define _ALPHA_STAT_H
//32 位的索引节点表的字段结构体定义
struct stat {
        unsigned int    st_dev;         //文件所在位置的设备号
        unsigned int    st_ino;         //文件的索引节点号
        unsigned int    st_mode;        //文件的类型
        unsigned int    st_nlink;       //连接到该文件的其他文件数量
        unsigned int    st_uid;         //文件所属用户
        unsigned int    st_gid;         //文件所属用户所在组
        unsigned int    st_rdev;        //如果是设备文件，则保存设备号，否则无效
        long            st_size;        //文件长度，如果是设备文件则为 0
        unsigned long   st_atime;       //最近一次访问文件的时间
        unsigned long   st_mtime;       //最近的修改文件时间
        unsigned long   st_ctime;       //最近一次对文件状态进行修改的时间
        unsigned int    st_blksize;     //文件系统的块大小
        unsigned int    st_blocks;      //文件所分配的块数
        unsigned int    st_flags;       //文件的用户定义标志
        unsigned int    st_gen;         //文件产生编号
};
//以下是 64 位系统的一些关于索引节点表的定义，增加了一些项
//修改了一些项，可以参考上一个结构体
```

```
struct stat64 {
        unsigned long   st_dev;
        unsigned long   st_ino;
        unsigned long   st_rdev;
        long            st_size;
        unsigned long   st_blocks;
        unsigned int    st_mode;
        unsigned int    st_uid;
        unsigned int    st_gid;
        unsigned int    st_blksize;
        unsigned int    st_nlink;
        unsigned int    _pad0;
        unsigned long   st_atime;
        unsigned long   st_atime_nsec;
        unsigned long   st_mtime;
        unsigned long   st_mtime_nsec;
        unsigned long   st_ctime;
        unsigned long   st_ctime_nsec;
        long            _unused[3];
};
#endif
```

5.1.3 Linux 的文件和文件属性

Linux 的文件是一个简单的字节数据序列，所以在 Linux 下对于文本文件、二进制文件的结构和访问方法是一样的。Linux 的文件是由一系列块（block）组成，每个块可能含有 512、1024、2048 或 4096 个字节，具体由系统实现决定，在同一个文件系统中块大小是相同的。当使用较大块的时候，由于每次磁盘操作可以传输更多的数据，操作所花的时间较少，所以可以提高磁盘和内存间数据的传输率，但是由于块太大，存储的有效容量将会下降，也就是说会浪费一些存储空间。

文件系统对文件的管理不仅仅是结构上的，同时还对文件属性进行了说明和管理，文件的属性包括：文件类型、文件长度、文件所有者、文件的许可权、文件最后的修改时间等。用户可以设置目录和文件的权限，以便允许或拒绝其他人对其进行访问。

在终端的根目录下使用“ls –l”命令，可以看到相应的文件属性说明如下：

```
alloy@ubuntu:/$ ls -l
总用量 54
drwxr-xr-x     2 root root    5120    1 月    8 15:24 bin
drwxr-xr-x     3 root root    3072    2 月   13 18:22 boot
drwxr-xr-x    16 root root    4160    2 月   20 10:16 dev
drwxr-xr-x   145 root root    8192    2 月   20 10:12 etc
drwxr-xr-x     3 root root    1024    1 月   22   2013 home
drwxrwxrwx     1 root root    4096    2 月   19 00:30 host
lrwxrwxrwx     1 root root      33    1 月    6 16:53 initrd.img -> /boot/initrd.img-3.8.0-35-generic
lrwxrwxrwx     1 root root      33   12 月   21 22:18 initrd.img.old -> /boot/initrd.img-3.8.0-34-generic
```

```
……//此处省略部分
drwxr-xr-x   11 root root    1024 12 月  22 10:50 usr
drwxr-xr-x   13 root root    1024   2 月  19 16:01 var
lrwxrwxrwx     1 root root       29    1 月    6 16:53 vmlinuz -> boot/vmlinuz-3.8.0-35-generic
lrwxrwxrwx     1 root root       29 12 月  21 22:18 vmlinuz.old -> boot/vmlinuz-3.8.0-34-generic
```

从上面可以看到文件类型、文件属性、用户名、用户所在组、文件大小、修改时间、文件名等信息，其中类似“drwxr-xr-x”的项说明了文件的类型和属性，其包含了 10 位字符，可以分为 4 组，如图 5.5 所示。

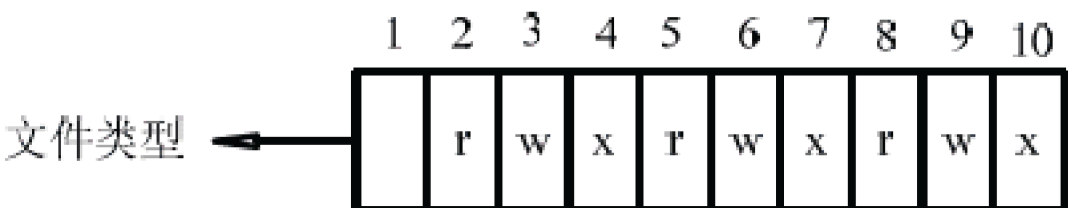


图 5.5 文件类型和属性

对文件类型和属性说明如下。

- 第 1 组：第 1 位，表示文件的类型，包括了普通文件、目录文件、管道文件等。
- 第 2 组：2~4 位，表示文件所有者（User）的权限，分别为读、写、执行。
- 第 3 组：5~7 位，表示文件所有者的同组用户（Group）的权限，分别为读、写、执行。
- 第 4 组：8~10 位，表示其他组用户（Other）权限，同样分别为读、写、执行。

第 1 位（组）是文件的类型说明，其由 stat 结构体中的 st_mode 来决定，标志符和对应的文件以及在 stat 中定义的关键字如表 5.1 所示。

表 5.1 文件类型说明

类型	说明
-	普通文件，对应屏蔽位关键字 S_ISEEG
l	链接文件，对应屏蔽位关键字 S_ISLNK
c	字符设备文件，对应屏蔽位关键字 S_ISCHR
s	套接字文件，对应屏蔽位关键字 S_ISSOCK
d	目录文件，对应屏蔽位关键字 S_SDIR
b	块设备文件，对应屏蔽位关键字 S_ISBLK
p	管道文件，对应屏蔽位关键字 S_ISFIFO

第 2~4 组的 2~10 位分别是对文件权限的说明，其中三位为一组，组中每一位以“1”来指示允许该操作（读、写、执行），以“0”来指示不允许该操作，所以每组中有“000”、“001”、“111”共 8 种不同的二进制编码组合，将每组的二进制编码转换为十进制即可得到“0”～“7”共 8 个不同数值，分别代表该组的一种操作组合，所以三个组共有 27 种不同的组合形式，例如“777”代表对文件的所有者、文件所有者的同组用户和其他组用户都对该文件拥有读、写和执行权限。

5.2　Linux 文件属性的操作方法

本节将介绍使用 C 语言对 Linux 的文件属性进行操作的方法，包括获取文件的时间和状态参数、获取文件的访问权限、修改文件的名称等。

5.2.1　判断文件类型

在第 5.1.2 小节中已经介绍了 Linux 在 stat.h 头文件中使用了一个结构体 stat 来存放文件的相应属性，可以使用 stat、fstat 和 lstat 函数获得文件的属性结构体，如果获取成功则返回 0，否则返回-1。

对 stat、fstat 和 lstat 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *pathname, struct stat *sbuf);
int fstat(int fd, struct stat *sbuf);
int lstat(const char *pathname,, struct stat *sbuf);
```

stat 函数和 lstat 函数使用文件的路径作为参数来标识需要获取文件类型的文件，而 fstat 函数使用文件对应的描述符来标识需要获取类型的文件；lstat 函数和 stat 函数的区别在于如果目标文件是一个符号链接，则 lstat 返回的是该符号链接的有关信息，而 stat 返回的是符号链接所引用的文件信息。


注　意

可以简单地把符号链接和符号链接的对应文件关系理解为 Windows 中的快捷方式和快捷方式对应的文件。

对 stat 系列函数的各个参数说明如下。

- pathname：目标文件的路径，可以是绝对路径或者相对路径，在 stat 和 lstat 函数中使用。
- fd：目标文件的文件描述符，通常由其他函数返回，在 fstat 函数中使用。
- sbuf：指向存放目标文件状态结构体的目标指针。

【例 5.1】使用 stat 函数获得指定文件的属性

例 5.1 是一个使用 stat 函数来获得 argv 指定文件类型的实例，应用代码首先使用 lstat 函数获取 argv 指定文件的属性参数，并且将其放入 stat_buf 结构体中，使用屏蔽码 S_IFMT 对 st_mode 中的文件类型进行屏蔽后再在 switch 语句中对屏蔽结果进行判断，然后输出相应的结果，其流程如图 5.6 所示。

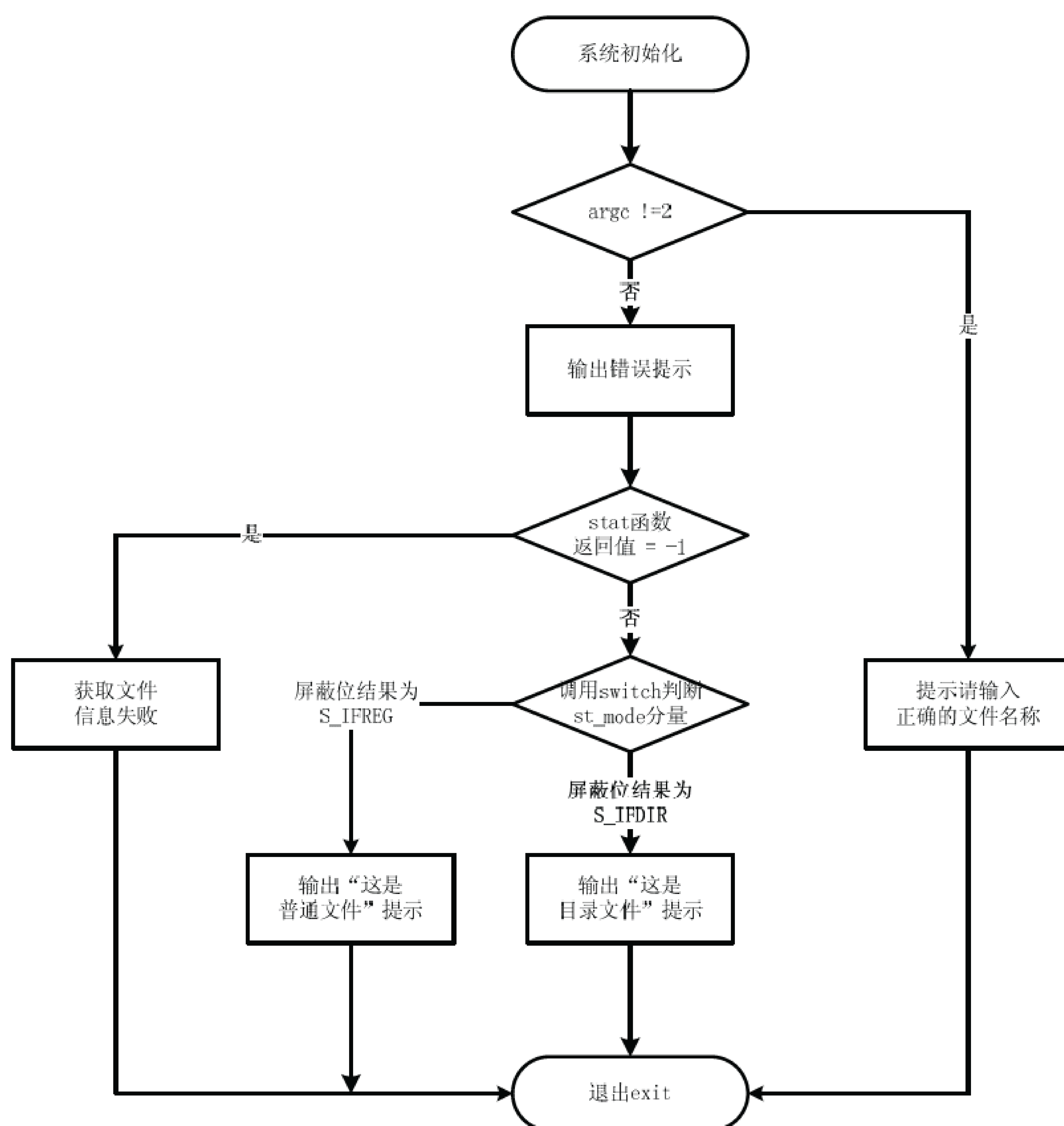


图 5.6 使用 stat 函数获取文件属性

实例的应用代码如下：

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <sys/stat.h>
5  int main(int argc, char *argv[])
6  {
7     int ret;
8     struct stat stat_buf;                    //定义 stat 结构体变量
9     if(argc != 2)                            //检查命令行参数
10    {
11       printf("请输入正确的文件参数！ \n");
12       return 0;
13    }
14    ret = stat(argv[1], &stat_buf);          //获取文件属性
15    if(ret == -1)                            //获取文件属性失败
16    {
17       perror("获取文件属性失败！ \n");
```

```
18          exit(0);
19      }
20      switch(stat_buf.st_mode & S_IFMT)
21      //判断文件类型，S_IFMT 是 st_mode 中文件类型的屏蔽码
22      {
23          case S_IFDIR:                                   //目录文件
24          printf("这是一个目录文件！\n");
25          break;
26          case S_IFREG:                                   //普通文件
27          printf("这是一个普通文件！\n");
28          break;
29      }
30      return 0;
31  }
```

将文件保存为 exam501stat.c，使用 gcc 对其进行编译链接，生成可执行文件 exam501stat。

```
alloy@ubuntu:~/linuxc/chapter5$ gcc exam501stat.c -o exam501stat
```

使用 ls 命令列举当前目录下的文件，其中 stattestdir 是一个目录文件。

```
alloy@ubuntu:~/linuxc/chapter5$ ls
exam501stat   exam501stat.c   exam502lstat.c   exam50xlink   exam50xlink.c   stattestdir
```

分别对普通文件 exam50xlink 和目录文件 statestdir 调用刚刚编译生成的可执行文件 exam501stat，可以看到如下的输出。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam501stat exam50xlink
这是一个普通文件！
alloy@ubuntu:~/linuxc/chapter5$ ./exam501stat stattestdir/
这是一个目录文件！
```

例 5.1 仅仅是对目录文件和普通文件进行判断，例 5.2 是一个给出了更多类型文件判断的实例，在该实例中使用了 lstat 函数来替代 stat 函数获得文件的属性，对于符号链接文件而言，lstat 函数返回的是该符号链接的信息而不是该符号链接引用的文件信息，所以如果当 Linux 的文件系统目录进行操作时必须使用 lstate 命令而不是 state 命令。

【例 5.2】使用 lstat 函数获得指定文件的属性

本实例和例 5.1 的区别除了使用 lstate 函数替代 stat 函数之外，还支持同时对多个文件的属性进行判断，并且使用 if 语句替代了 switch 语句。

实例的应用代码如下：

```
1  //这是一个使用 lstat 函数来获得 arvg 指定文件属性的实例
2  #include <fcntl.h>
3  #include <sys/stat.h>
4  #include <stdio.h>
5  int main(int argc, char *argv[])
6  {
```

```
7       int i;
8       struct stat buf;                      //文件属性存放缓冲区
9       char *ptr;
10      for (i=1; i<argc;i++)                 //命令行参数作为输出参数
11      {
12          printf("%s 是一个 ", argv[i]);      //首先打印输出文件名
13          if (lstat(argv[i], &buf) < 0)
14          //如果 lstat 函数返回值小于 0，则表示函数调用失败
15          {
16              printf("lstat error");
17              continue;                     //仅仅退出当前循环
18          }
19          //以下开始判断文件类型
20          if (S_ISREG(buf.st_mode))
21          {
22               ptr = "普通文件";              //普通文件
23          }
24          else if (S_ISDIR(buf.st_mode))
25          {
26                ptr = "目录文件";             //目录文件
27          }
28          else if (S_ISCHR(buf.st_mode))
29          {
30                ptr = "字符设备文件";         //字符设备文件
31          }
32          else if (S_ISBLK(buf.st_mode))
33          {
34                ptr = "块设备文件";           //块设备文件
35          }
36          else if (S_ISFIFO(buf.st_mode))
37          {
38                ptr = "FIFO";                 //先进先出文件
39          }
40          else if (S_ISLNK(buf.st_mode))
41          {
42                ptr = "符号链接";             //符号链接文件
43          }
44          else if (S_ISSOCK(buf.st_mode))
45          {
46                ptr = "套接字文件";           //套接字文件
47          }
48          else    //如果不是以上文件类型，则表明为未知文件类型
49          {
50              ptr = "未知文件类型";
51          }
52          printf("%s\n", ptr);              //输出文件类型
53      }
54      return 0;
55  }
```

将文件保存为 exam502lstat.c，使用 gcc 对其进行编译链接，生成可执行文件 exam502lstat。

```
alloy@ubuntu:~/linuxc/chapter5$ gcc exam502lstat.c -o exam502lstat
```

调用 exam502lstat 获取当前目录下的 exam501stat.c 文件类型。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam502lstat exam501stat.c
exam501stat.c 是一个 普通文件
```

调用 exam502lstat 获取非当前目录下文件的类型，linuxc 是根目录下当前用户的一个目录文件。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam502lstat /home/alloy/linuxc
/home/alloy/linuxc 是一个 目录文件
```

前面提到过 exam502lstat 支持一次获取多个文件的属性，可以同时获取当前目录下的 exam502lstat.c 和目录文件 stattestdir 的类型。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam502lstat exam502lstat.c stattestdir/
exam502lstat.c 是一个 普通文件
stattestdir/ 是一个 目录文件
```

5.2.2 文件的时间信息

stat 系列函数返回的是文件的属性状态，而如果要对文件相应的时间信息进行操作，可以使用 utime 函数，如果调用成功将返回 0，并且自动更新文件的特性修改时间 st_ctime，否则返回-1。

在 Linux 系统中，每个文件都有三个对应的时间信息，如表 5.2 所示，其分别对应 stat 结构体中如下三个字段。

```
        unsigned long    st_atime;   //最近一次访问文件的时间
        unsigned long    st_mtime;   //最近的修改文件时间
        unsigned long    st_ctime;   //最近一次对文件状态进行修改的时间
```

表 5.2 文件的时间信息

文件字段	说明	对应的操作函数
st_atime	文件的最后访问时间	read
st_mtime	文件的最后修改时间	write
st_ctime	文件索引节点（innode）的最后修改时间	chmod

st_atime 和 st_ctime 这两个时间的主要区别是前者是最后一次对文件本身进行修改操作的时间，而后者是对文件的索引节点 innode 进行操作的时间；前者受到相应的函数例如 write 的影响，后者的改变则不一定要涉及对文件内容的操作，只需要修改文件的状态，例如文件的访问权限等就会产生。



注 意

可以利用 utime 函数来改变一个文件的访问时间和修改时间，但是没有函数可以改变文件的特性修改时间，因为其是由系统来维护的。

对 utime 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *pathname,const struct utimebuf *times );
```

对 utime 函数的各个参数说明如下。

- pathname：目标文件的路径参数。
- times：用于存放 utime 返回的时间信息，其是 utime 函数使用的一个数据结构，对其说明如下：

```
struct utimbuf
{
      time_t actime;
      time_t modtime;
}
```

- actime：文件的访问时间。
- modtime：文件的修改时间。

需要注意的是这两个时间值都是日历时间，也就是自标准时间（1970 年 1 月 1 日 00:00:00）起到当前所经过的秒数。

如果 times 是空指针，文件的访问时间和修改时间均设置为当前时间，此时，要么进程的有效用户 ID 必须等于文件的用户 ID，要么进程必须拥有该文件的写权限。

如果 times 不是空指针，它可解释为指向 utimebuf 结构的指针，并且用 times 值更新文件的访问时间和修改时间。在这种情形下，要么进程的有效用户 ID 必须等于文件的用户 ID，要么必须是超级进程，仅具有文件的写权限是不够的。

【例 5.3】使用 utime 函数操作文件的时间参数

例 5.3 是一个使用 utime 函数对文件的时间参数进行修改的实例，文件首先使用 stat 函数获得文件当前的时间参数，然后使用 open 函数对文件进行修改，再用 utime 函数对文件的时间信息进行修改，文件名由 argv 参数给出，其流程如图 5.7 所示。

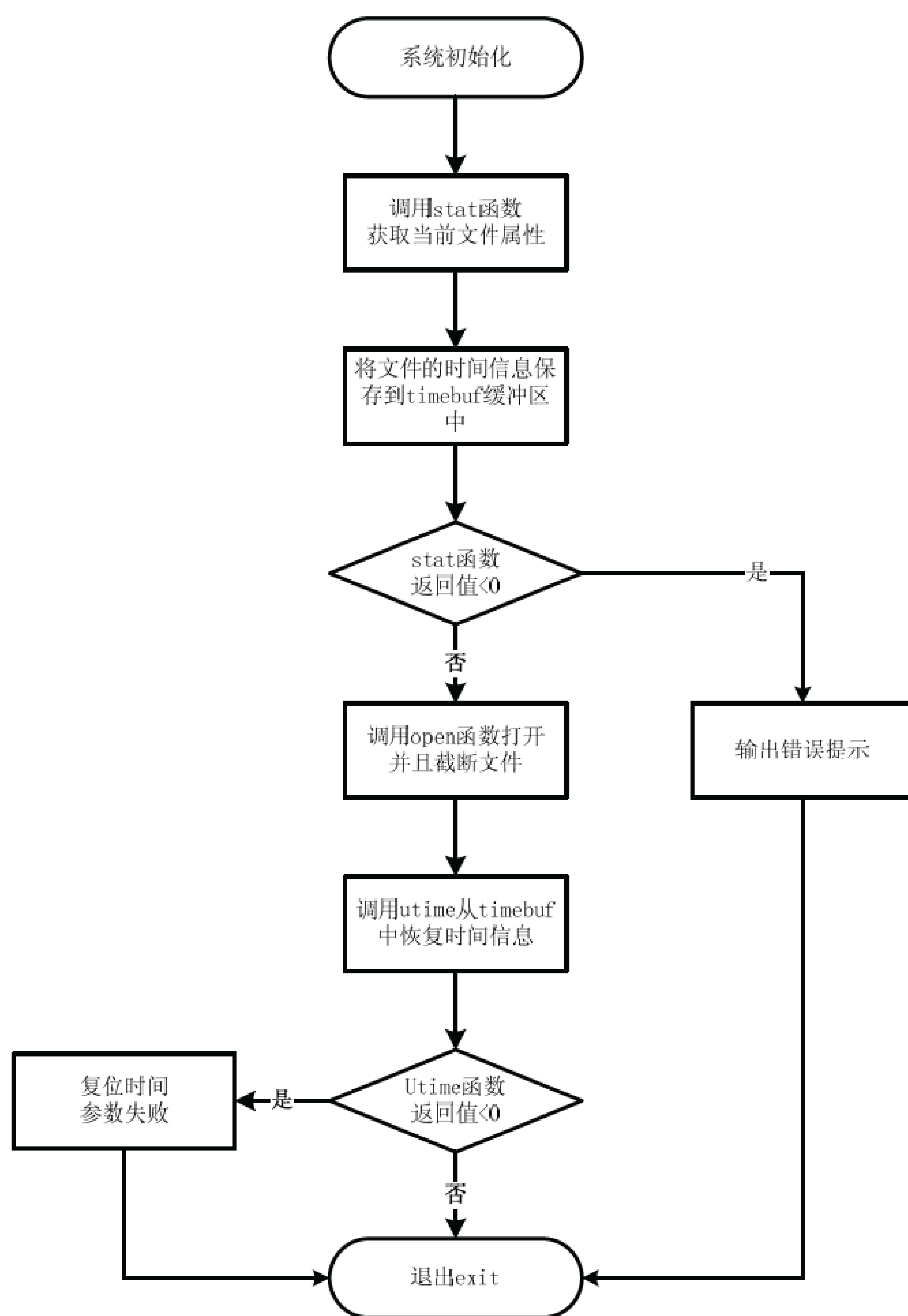


图 5.7 使用 utime 操作文件的时间参数

实例的应用代码如下：

```
1   //这是一个使用 utime 函数对文件的时间参数进行修改的实例
2   //文件首先使用 stat 函数获得文件当前的时间参数，然后使用
3   //Open 函数对文件进行修改，再用 utime 函数对文件的时间信息
4   //进行修改，文件名由 argv 参数给出
5   #include <stdio.h>
6   #include <fcntl.h>
7   #include <utime.h>
8   int main(int argc, char *argv[])
9   {
10          int i, fd;
11          struct stat statbuf;            //文件信息缓冲区
12          struct utimbuf timebuf;         //文件时间信息缓冲区
13          for (i=1;i<argc;i++)            //支持同时对多个文件进行操作
14          {
```

```
15            if(stat(argv[i], &statbuf) < 0)                   //获得文件当前信息失败
16            {
17                printf("获取文件信息失败\n");                  //输出提示并且进行到下一个文件
18                continue;
19            }
20        if((fd = open(argv[i], O_RDWR|O_TRUNC)) < 0)      //尝试打开并且截断文件
21        {
22            printf("打开截断文件操作失败\n");                  //文件打开失败
23            continue;
24        }
25        close(fd);    //关闭文件
26        timebuf.actime     = statbuf.st_atime;
27        timebuf.modtime = statbuf.st_mtime;                     //恢复时间
28        if (utime(argv[i], &timebuf) < 0)                      //复位时间失败
29        {
30            printf("时间操作失败\n");                          //复位时间失败
31            continue;
32        }
33    }
34    return 0;
35  }
```

将文件保存为 exam503utime.c，在终端中使用 gcc 编译，生成可执行文件 exam503utime。

```
alloy@ubuntu:~/linuxc/chapter5$ gcc exam503utime.c -o exam503utime
```

使用 vim 在当前目录下创建一个 utimetest.txt 文件，在其中输入任意字符串，然后保存，调用“ls –l”命令查看该文件的属性，此时可以看到操作前文件的长度为 44，最后修改时间为 2 月 20 日的下午 18 点 08。

```
alloy@ubuntu:~/linuxc/chapter5$ ls -l utimetest.txt
-rw-rw-r-- 1 alloy alloy 44    2 月  20 18:08 utimetest.txt
```

调用“ls –lu”命令查看该文件的最后访问时间，可以看到最后访问时间和最后修改时间相同。

```
alloy@ubuntu:~/linuxc/chapter5$ ls -lu utimetest.txt
-rw-rw-r-- 1 alloy alloy 44    2 月  20 18:08 utimetest.txt
```

调用 date 命令显示当前的 Linux 系统时间。

```
alloy@ubuntu:~/linuxc/chapter5$ date
2014 年  02 月  20 日  星期四  18:11:42 CST
```

调用 exam503utime 可执行文件对 utimetest.txt 文件进行操作，可以看到没有错误信息出现，表示操作成功。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam503utime utimetest.txt
```

再次调用“ls –l”、“ls –lu”命令查看该文件的属性，可以看到文件的最后访问时间和修改时没

有发生变化，但是文件的长度变成了 0，此时再调用“ls –lc”命令查看文件的状态修改时间变成了 2 月 20 日的 18 点 12 分，即为调用 exam503utime 可执行文件的时间。

```
alloy@ubuntu:~/linuxc/chapter5$ ls -l utimetest.txt
-rw-rw-r-- 1 alloy alloy 0   2 月  20 18:08 utimetest.txt
alloy@ubuntu:~/linuxc/chapter5$ ls -lu utimetest.txt
-rw-rw-r-- 1 alloy alloy 0   2 月  20 18:08 utimetest.txt
alloy@ubuntu:~/linuxc/chapter5$ ls -lc utimetest.txt
-rw-rw-r-- 1 alloy alloy 0   2 月  20 18:12 utimetest.txt
```

文件的各种相关操作函数对文件的最后访问时间、最后修改时间和最后更改时间的影响说明如表 5.3 所示，“×”代表无影响，“●”表示有影响。

表 5.3　文件操作函数对文件时间的影响

函数	函数作用	访问时间	修改时间	更改时间
chmod/fchmod	修改文件权限	×	×	●
create	创建文件	●	●	●
open	打开文件	●	●	●
read	读文件	●	×	×
write	写文件	×	●	●
rename	修改文件名称	×	×	●
remove	删除文件	×	×	●
link	创建符号链接文件	×	×	●
mkdir	创建目录文件	●	●	●
rmdir	删除目录文件	×	×	×
unlink	删除符号链接文件	×	×	●
utime	修改文件的时间参数	●	●	●

5.2.3　文件的权限

对于 Linux 中的文件来说，其必须为系统中的某个进程进行操作，而和一个进程相关的 ID 有 6 个或者更多，包括实际用户 ID、实际组 ID、有效用户 ID、有效组 ID、附加组 ID、保存的设置用户 ID 和保存的设置组 ID，对其说明如下：

- 实际用户 ID、实际组 ID：用于表示当前 Linux 的登录用户，其在登录时由口令文件中的登录项获得，通常来说在整个会话过程中这个登录用户并不会改变，但是如果在 Ubuntu 等环境下使用 sudo 类似的命令可以暂时改变。
- 有效用户 ID、有效组 ID 和附加组 ID：用于决定每个文件的访问权限，其保存在 stat 结构体的 st_mode 分量中，如表 5.4 所示，该表和图 5.5 中的第 2~4 组是对应的。

表 5.4　文件的访问权限位

st_mode 分量	意义
S_IRUSR	用户读
S_IWUSR	用户写
S_IXUSR	用户执行
S_IRGRP	组读
S_IWGRP	组写
S_IXGRP	组执行
S_IROTH	其他读
S_IWOTH	其他写
S_IXOTH	其他执行

- 保存的设置用户 ID 和保存的设置组 ID：保存了包含有效用户 ID 和有效组 ID 的一个副本，在后续章节中将进行详细介绍。

通常来说，有效用户 ID 和实际用户 ID 是相同的，而有效组 ID 和实际组 ID 也是相同的，每个文件都有一个所有者和一个组所有者，所有者存放在 stat 结构中的 st_uid 分量中，而组所有者存放在 st_gid 分量中。



在第 3 章的 open 函数、create 函数等示例中，并没有涉及新建文件的用户 ID 和组 ID 操作，此时 Linux 内核以当前创建文件进程的有效用户 ID 和有效组 ID 作为文件的相应用户 ID 和组 ID 进行操作。

文件的用户 ID 和组 ID 与当前进程的用户 ID 和组 ID 未必相同，此时如果希望对文件进行操作，则需要先测试该文件的相应权限，此时可以使用 access 函数，当测试成功（拥有相应权限）时返回 0，否则返回-1。

对 access 函数的标准调用格式说明如下：

```
#include <unistd.h>
int access(const char *pathname,int mode );
```

对 access 函数的各个参数说明如下。

- pathname：目标文件的路径，可以是绝对路径或者相对路径。
- mode：用于标识当前检查的文件权限类比，需要注意的是这个参数是不能采用“|”来连接这些参数的，只能选择如表 5.5 所示的 4 种参数中的其中一个作为当前 access 函数的 mode 参数。

表 5.5　access 函数中 mode 的 4 种参数

参数	说明
R_OK	检验调用进程是否有读访问权限
W_OK	检验调用进程是否有写访问权限
X_OK	检验调用进程是否有执行访问权限
F_OK	检验规定的文件是否存在

【例 5.4】使用 access 函数测试文件的权限

例 5.4 是一个使用 access 函数对指定文件的读、写、运行等属性进行测试的实例，文件的名称由 argv 参数给出，应用代码首先测试文件是否存在，然后依次调用 access 函数分别测试文件的读、写、执行权限，其执行流程如图 5.8 所示。

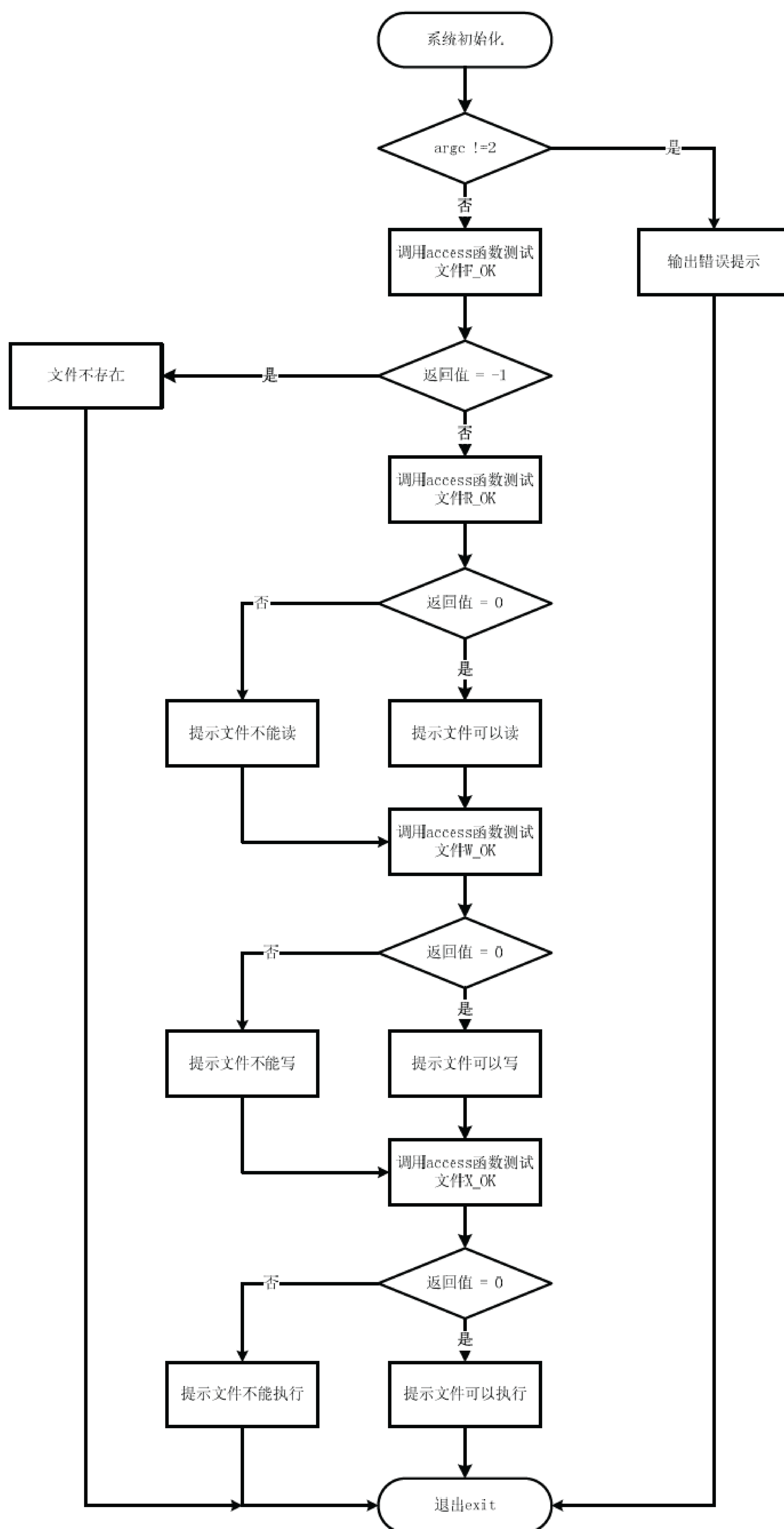


图 5.8　使用 access 函数测试文件的权限

实例的应用代码如下：

```
1  //对 argv 传递文件名的文件使用 access 文件进行权限检测
2   #include <fcntl.h>
3   #include <stdio.h>
4   int main(int argc,char *argv[])
5   {
6      int temp;
7      if(argc != 2)                          //参数错误
8      {
9         printf("文件名参数错误!\n");
10        return 1;
11     }
12     temp = access(*(argv+1),F_OK);         //测试文件是否存在
13     if(temp == -1)
14     {
15        printf("文件不存在!\n");             //文件不存在
16        return 2;
17     }
18     temp = access(*(argv+1),R_OK);         //测试文件是否能读
19     if(temp == 0)
20     {
21        printf("%s 文件可以进行读操作!\n",*(argv+1));
22     }
23     else
24     {
25        printf("%s 文件不允许进行读操作!\n",*(argv+1));
26     }
27     temp = access(*(argv+1),W_OK);         //测试文件是否能写
28     if(temp == 0)
29     {
30         printf("%s 文件可以进行写操作!\n",*(argv+1));
31     }
32     else
33     {
34         printf("%s 文件不允许进行写操作!\n",*(argv+1));
35     }
36     temp = access(*(argv+1),X_OK);         //测试文件是否可执行
37     if(temp == 0)
38     {
39         printf("%s 是一个可执行文件!\n",*(argv+1));
40     }
41     else
42     {
43         printf("%s 不是一个可执行文件!\n",*(argv+1));
44     }
45     return 0;
46  }
```

将文件保存为 exam504access.c，在终端中使用 gcc 编译链接生成可执行文件 exam504access。

```
alloy@ubuntu:~/linuxc/chapter5$ gcc exam504access.c -o exam504access
```

使用 exam504access 测试例 5.3 所生成的可执行文件 exam503utime，可以看到这是一个可以进行读、写、执行的文件。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam504access exam503utime
exam503utime 文件可以进行读操作!
```

```
exam503utime 文件可以进行写操作!
exam503utime 是一个可执行文件!
```

使用 exam504access 测试例 5.3 的 C 语言文件 exam503utime.c，其可以进行读写操作，但是不能执行。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam504access exam503utime.c
exam503utime.c 文件可以进行读操作!
exam503utime.c 文件可以进行写操作!
exam503utime.c 不是一个可执行文件!
```

文件的权限可以使用 access 进行测试，在创建一个文件的时候，可以使用文件创建屏蔽字 umask 来设置该文件的相应权限。文件创建屏蔽字与文件权限字一样都是一个位串，并且与文件权限位一一对应。每当进程创建一个新文件或新目录时，它所指定的文件访问权限位将受到文件创建屏蔽字 umask 的影响以确定其权限。

例如当 umask 的值为 007（即其他用户的所有访问权限位为 1）时，则表示新创建文件的其他用户访问权限位均为 0，即其他用户没有任何访问权限，即使创建函数中指定 mode 参数允许这种权限也不例外。

表 5.6 是以八进制给出的 umask 文件创建屏蔽字说明。

表 5.6　umask 文件创建屏蔽字说明

umask 值	说明	umask 值	说明
0400	用户读	0200	用户写
0100	用户执行	0040	组读
0020	组写	0010	组执行
0004	其他读	0002	其他写
0001	其他执行		

在 Shell 中可以使用 umask 命令来获得当前的 umask 值，也可以修改这个值，例如当前（进程）的 umask 值如下，其对应的是“其他写”。

```
alloy@ubuntu:~/linuxc/chapter5$ umask
0002
```

如果在 umask 命令后加入“-S”参数，则可以以符号形式输出当前的 umask 值，其中 u 表示当前用户，g 表示当前用户组，o 表示其他用户，可以看到当前的 umask 值对应的是支持当前用户读（r）、写（w）和执行（x）操作，支持当前用户组读（r）、写（w）和执行（x）操作，支持其他用户的读（r）和执行（x）操作。

```
alloy@ubuntu:~/linuxc/chapter5$ umask -S
u=rwx,g=rwx,o=rx
```

可以使用 umask 命令来修改当前的 umask 值。

```
alloy@ubuntu:~/linuxc/chapter5$ umask 0010
alloy@ubuntu:~/linuxc/chapter5$ umask
0010
```

在 Linux 的 C 语言代码中，可以使用 umask 函数在创建文件的时候指定该文件的权限，对 umask 函数的标准调用格式说明如下，其返回值是修改之前的文件屏蔽字，其参数 cmask 是新设定的文件屏蔽字值，其实质上是表 5.5 中 9 个常量中一个或者几个按位或的结果。

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

【例 5.5】使用 umask 函数指定新建文件权限

例 5.5 是使用 umask 函数来指定两个新建文件权限的实例，应用代码首先使用 umask 指定新建文件 1 的权限为当前默认权限，调用 create 函数来创建文件 umasktest1，然后使用 umask 修改当前文件屏蔽字的属性为“S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH（组用户读|组用户写|其他用户读|其他用户写）”，再次调用 create 函数创建文件 umasktest2，其流程如图 5.9 所示。

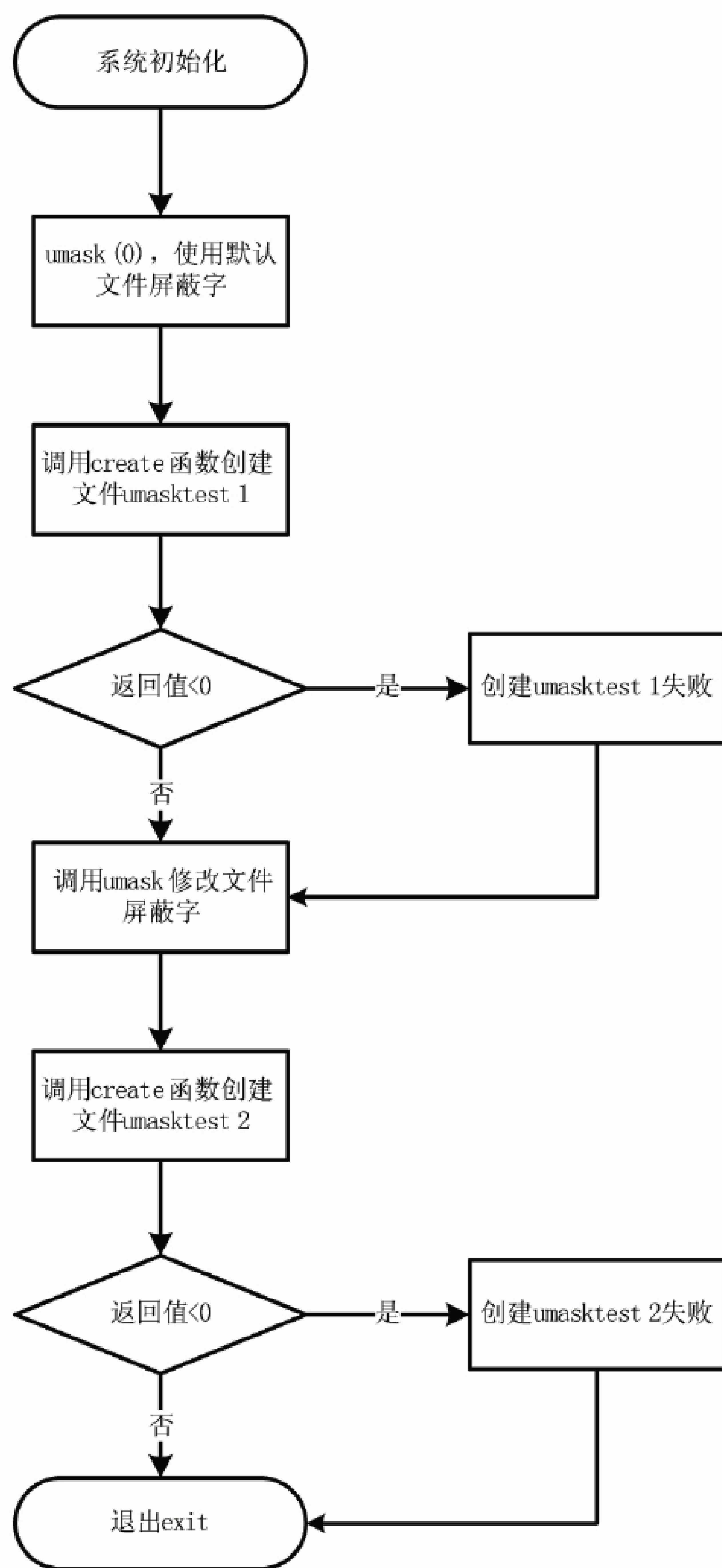


图 5.9　使用 umask 函数指定新建文件权限

实例的应用代码如下：

```
1   //使用 umask 函数来修改文件属性关键字，并且创建两个测试文件
2   #include <fcntl.h>
3   #include <stdio.h>
4   #define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
5   //定义操作权限字符串
6   int main(void)
7   {
8        umask(0);      //原始默认权限
9        if (create("umasktest1", RWRWRW) < 0)      //创建文件 umasktest1
10       {
11           printf("创建测试文件 1 失败\n");
12       }
13       umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
14       //修改文件创建关键字
15      if (create("umasktest2", RWRWRW) < 0)      //创建文件 umasktest2
16      {
17           printf("创建测试文件 2 失败\n");
18      }
19      return 0;
20  }
```

将文件保存为 exam505umask.c，在终端中使用 gcc 编译链接，生成可执行文件 exam505umask。

```
alloy@ubuntu:~/linuxc/chapter5$ gcc exam505umask.c -o exam505umask
```

首先使用 umask 命令查看当前的文件屏蔽字为 0002。

```
alloy@ubuntu:~/linuxc/chapter5$ umask
0002
```

执行 exam505umask 可执行文件，在当前目录下创建两个文件，并且使用“ls –l”命令查看文件的对应属性。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam505umask
alloy@ubuntu:~/linuxc/chapter5$ ls -l umasktest1
-rw-rw-rw- 1 alloy alloy 0   2 月  20 23:57 umasktest1
alloy@ubuntu:~/linuxc/chapter5$ ls -l umasktest2
-rw------- 1 alloy alloy 0   2 月  20 23:57 umasktest2
```

umask 函数只有在文件创建时才能确定文件的相应权限，如果需要修改一个已经存在的文件权限，用户可以使用 chmod 函数或者 fchomd 函数，前者可以对任何指定文件进行操作，而后者必须以文件描述符对一个已经打开的文件进行操作。

对 chmod 和 fchmod 函数的标准调用格式说明如下，若操作成功则返回 0，否则返回-1。

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char * filename, mode_t mode);
int fchmod(int fd, mode_t mode);
```

对 chmod 和 fchmod 函数的各个参数说明如下。


注 意

在调用 chmod 函数或者 fchmod 函数的时候，进程的有效用户 ID 必须等于文件所有者的 ID，或者该进程必须具有超级用户的权限，关于进程权限的相关知识可以参考第 8 章。

- pathname：目标文件的路径，可以是绝对路径或者相对路径，在 chmod 函数中使用。
- fd：目标文件的文件描述符，在 fchmod 函数中使用。
- mode：和 umask 函数类似，chmod 和 fchmod 函数的 mode 参数也是表 5.5 中 9 个文件访问权限位的组合，对其说明如表 5.7 所示。

表 5.7 chmod/fchmod 函数的 mode 参数

mode 值	说明
S_ISUID	执行时设置用户 ID
S_ISGID	执行时设置组 ID
S_ISVTX	保存正文
S_IRWXU	用户读、写和执行，此为以下三项的“或”操作
S_IRUSR	用户读
S_IWUSR	用户写
S_IXUSR	用户执行
S_IRWXG	组读、写和执行，此为以下三项的“或”操作
S_IRGRP	组读
S_IWGRP	组写
S_IXGRP	组执行
S_IRWXG	其他读、写和执行
S_IROTH	其他读
S_IWOTH	其他写
S_IXOTH	其他执行

【例 5.6】使用 chmod 函数修改指定文件权限

例 5.6 是一个使用 chmod 函数修改两个指定文件权限的实例，应用代码首先使用 stat 函数取得 argv[1]指定文件的属性，然后使用 chmod 函数对该文件的权限进行修改，修改为(statbuf.st_mode & ~S_IXGRP) | S_ISGID 的运算结果，使用 chmod 函数直接对 argv[2]的属性进行修改，其流程如图 5.10 所示。

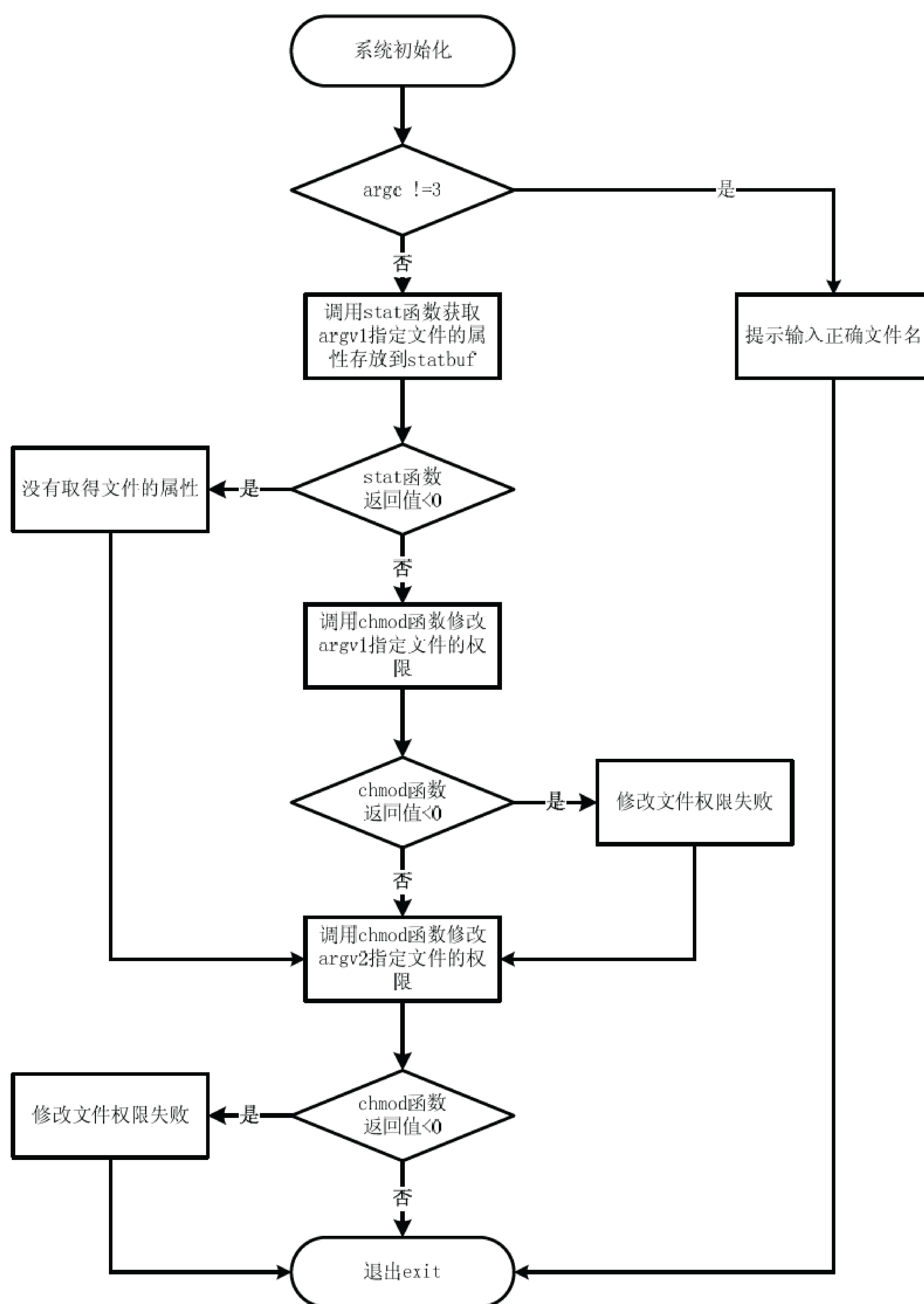


图 5.10　使用 chmod 函数修改文件权限

实例的应用代码如下：

```
1  //这是一个使用 chmod 函数来修改文件权限的实例
2  //文件名使用 argv 参数传递
3  #include <fcntl.h>
4  #include <stdio.h>
5  int main(int argc,char *argv[])
6  {
7        int ret;
8        struct stat statbuf;              //文件状态缓冲区
9        if(argc != 3)                     //如果参数格式错误
10       {
```

```
11            printf("请输入正确的 2 个文件名！ \n");
12            return 1;                              //直接退出
13        }
14        ret = stat(*(argv+1),&statbuf);            //获得文件的属性
15        if (ret< 0)                                //获取文件属性失败
16        {
17          printf("没有取得文件对应的属性!\n");
18        }
19        else
20        {
21          if(chmod(*(argv+1), (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
                                                     //修改参数 1 对应的文件权限
22          {
23              printf("修改文件%s 权限出错",*(argv+1));
24          }
25        }
26        if (chmod(*(argv+2), S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
                                                     //修改参数 2 对应文件权限
27        {
28            printf("修改文件%s 权限出错",*(argv+2));
29        }
30        return 0;
31  }
```

将文件保存为 exam506chmod.c，在终端中进行编译链接，生成 exam506chmod 可执行文件。

```
alloy@ubuntu:~/linuxc/chapter5$ gcc exam506chmod.c -o exam506chmod
```

调用 exam506chmod 可执行文件，修改例 5.5 中生成的 umasktest1 和 umasktest2 文件的权限，然后调用“ls –l”命令查看这两个文件的属性，可以对比例 5.5 中生成的文件属性。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam506chmod umasktest1 umasktest2
alloy@ubuntu:~/linuxc/chapter5$ ls -l umasktest1
-rw-rwSrw- 1 alloy alloy 0   2 月  20 23:57 umasktest1
alloy@ubuntu:~/linuxc/chapter5$ ls -l umasktest2
-rw-r--r-- 1 alloy alloy 0   2 月  20 23:57 umasktest2
```


注 意

如果需要修改文件的用户 ID 和用户组 ID，可以使用 chown、fchown 和 lchown 函数，读者可以自行参阅相应的手册。

5.2.4 修改文件的名称

在实际应用中，可能需要对一个文件进行改名操作，此时可以使用 rename 函数，对 rename 函数的标准调用格式说明如下，如果操作成功则返回 0，否则返回-1。

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

对 rename 函数的各个参数和应用实例说明如下。

- oldname：文件的旧文件名称，支持路径。
- newname：文件的新文件名称，同样支持路径。

需要注意的是，rename 函数既可以对文件进行操作，也可以对目录进行操作（前面说过，Linux 中的目录也是一个文件），对 rename 函数的参数说明如表 5.8 所示。

表 5.8　rename 函数参数总结

操作	newname 所示文件不存在	newname 指向普通文件	newname 指向目录文件
oldname 指向普通文件	文件被重命名	newname 被删除，原来名为 oldname 的文件被重命名为 newname	错误
oldname 指向目录文件	文件被重命名	错误	newname 所指向的目录文件为空目录，则该目录文件被删除，oldname 被重命名，否则出错

【例 5.7】使用 rename 函数修改文件名称

例 5.7 是一个使用 rename 函数来对 argv[1]指定的文件名称进行修改，将其修改为 argv[2]所指定的文件名，其流程如图 5.11 所示。

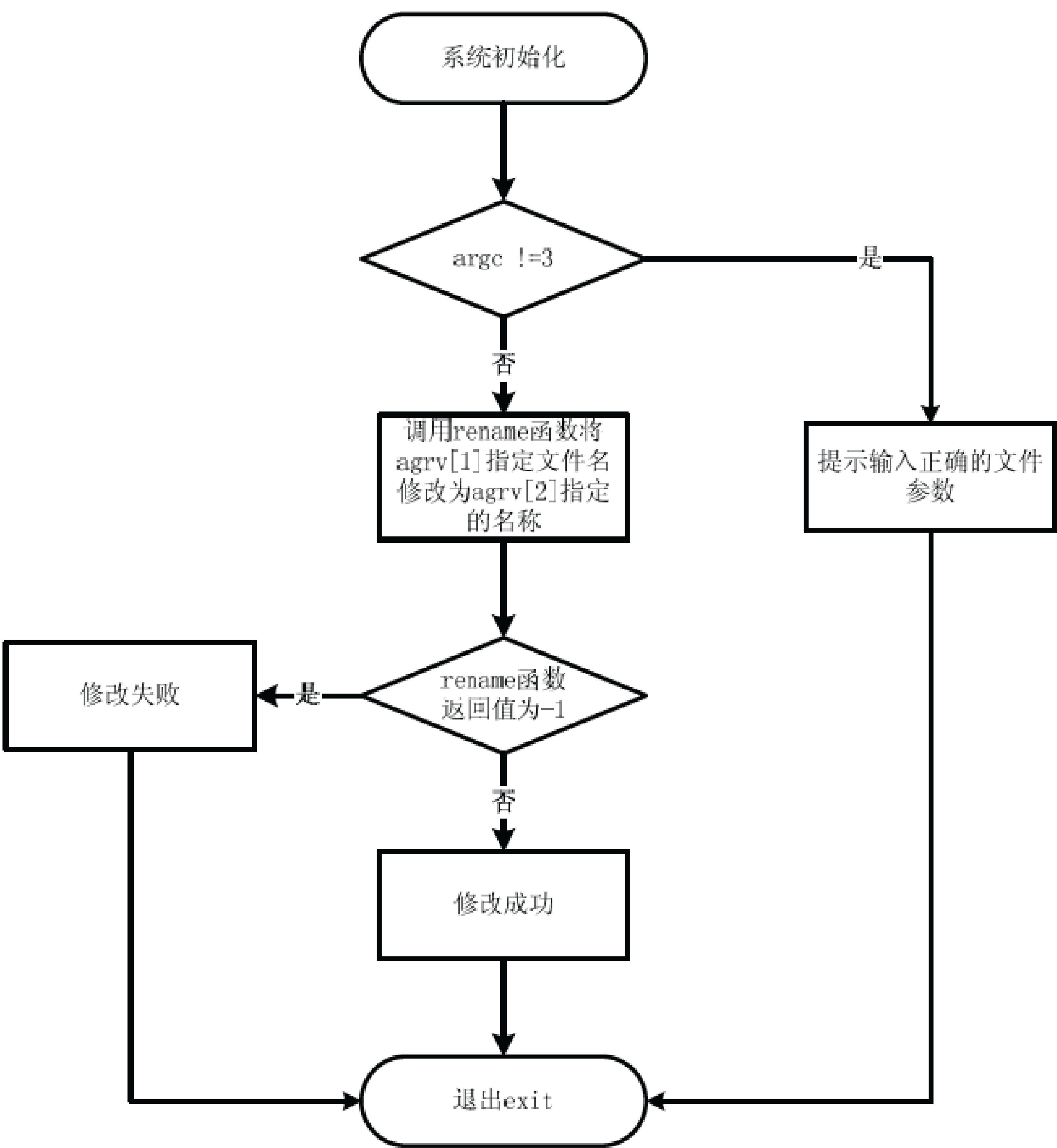


图 5.11　使用 rename 函数修改文件名称

实例的应用代码如下：

```
1  //这是一个将 argv1 给出的文件名称修改为 argv2 字符串的实例
2  #include <fcntl.h>
3  #include <stdio.h>
4  int main(int argc,char *argv[])
5  {
6     int temp;
7     if(argc != 3)                                  //如果不是三个参数，则报错
8     {
9        printf("文件参数错误！ \n");
10       return 1;
11    }
12    temp = rename(*(argv+1),*(argv+2));            //将前者修改为后者
13    if(temp == -1)                                 //如果修改文件出错
14    {
15       printf("修改%s 文件名失败！ \n",*(argv+1));   //改名出错
16    }
17    else
18    {
19       printf("将文件%s 名称修改为%s 成功！ \n",*(argv+1),*(argv+2));
20    }
21    return 0;                                      //退出
22 }
```

将文件保存为 exam507rename.c，在终端中进行编译链接，生成可执行文件 exam507rename。

```
alloy@ubuntu:~/linuxc/chapter5$ gcc exam507rename.c -o exam507rename
```

调用可执行文件 exam507rename 将例 5.5 生成的文件 umasktest2 名称修改为 umasktest3。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam507rename umasktest2 umasktest3
将文件 umasktest2 名称修改为 umasktest3 成功！
```

5.2.5　删除文件

在某些时候需要删除文件系统中的某个文件，此时可以调用 remove 函数，对其标准调用格式说明如下，如果调用成功则返回 0，否则返回-1。

```
#include <stdio.h>
int remove(const char *pathname);
```

remove 的参数 pathname 是需要删除文件的对应名称，

【例 5.8】使用 remove 函数删除指定文件

例 5.8 是一个使用 remove 函数删除 argv 参数指定文件的实例，其流程如图 5.12 所示。
实例的应用代码如下：

```
1   #include <stdio.h>
2   int main(int argc,char *argv[])
3   {
4       int ret;
5       if(argc != 2)                     //如果参数错误
6       {
7           printf("请输入待删除的文件名称！\n");
8           return 1;
9       }
10      ret = remove(*(argv+1));          //删除文件
11      if(ret == 0)                      //删除文件成功
12      {
13          printf("删除文件%s 成功！\n",*(argv+1));
14      }
15      return 0;
16  }
```

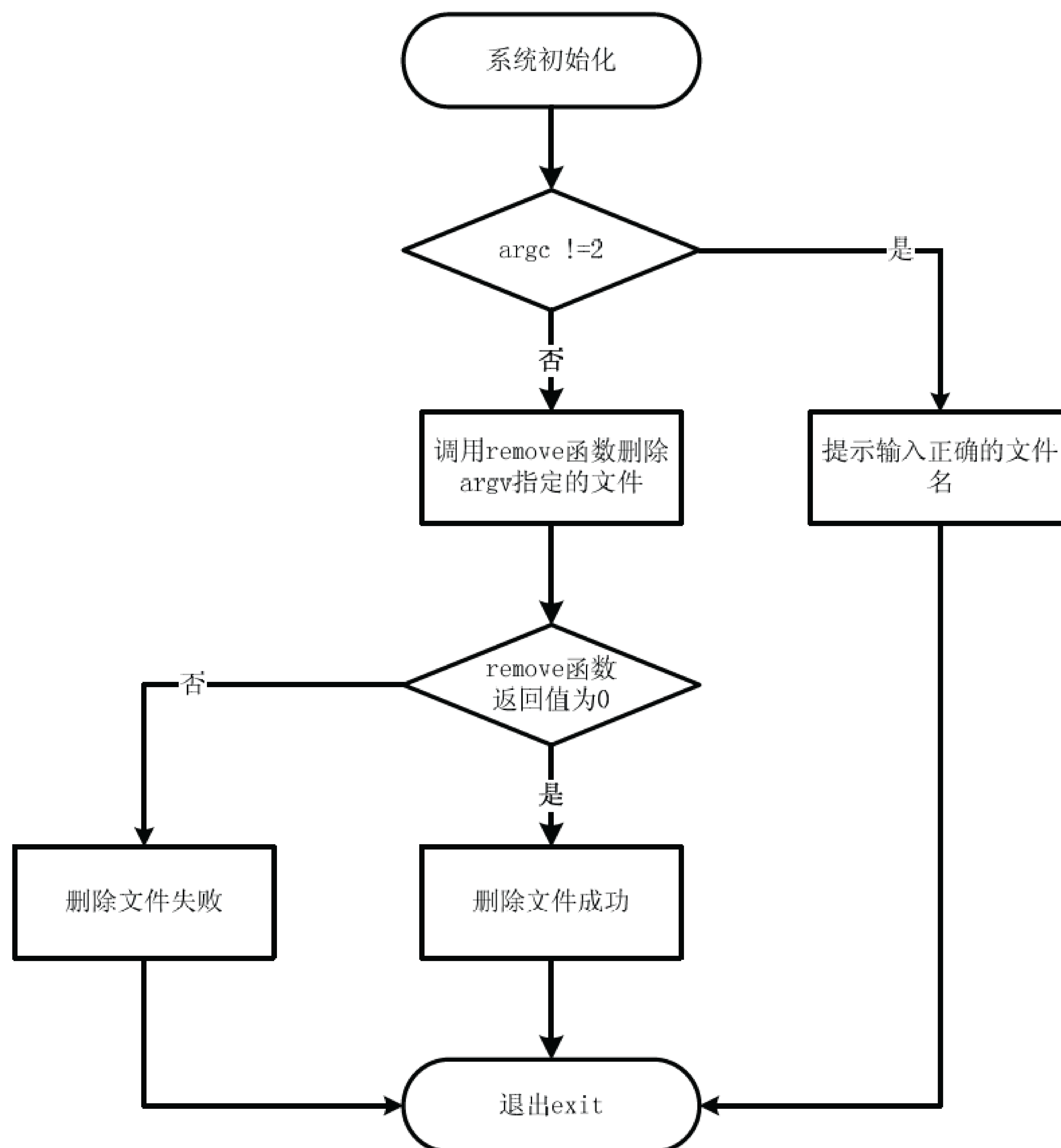


图 5.12　使用 remove 函数删除指定文件

将文件保存为 exam508remove.c，在终端中进行编译链接，生成 exam508remove 可执行文件。

```
alloy@ubuntu:~/linuxc/chapter5$ gcc exam508remove.c -o exam508remove
```

使用 vim 在当前目录下建立一个名称为 removetest.txt 的文件，在其中输入一些字符串并且保存，然后使用“ls –l”命令查看该文件的属性。

```
alloy@ubuntu:~/linuxc/chapter5$ vim removetest.txt
alloy@ubuntu:~/linuxc/chapter5$ ls -l removetest.txt
-rw-rw-r-- 1 alloy alloy 16   2 月  21 02:09 removetest.txt
```

调用 exam508remove 删除 removetest.txt 文件，执行过程如下。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam508remove removetest.txt
删除文件 removetest.txt 成功！
```

此时再次调用“ls –l”命令查看 removetest.txt 文件，会看到提示该文件不存在。

```
alloy@ubuntu:~/linuxc/chapter5$ ls -l removetest.txt
ls: 无法访问 removetest.txt: 没有那个文件或目录
```


注 意

还可以使用 remove 函数来解除对一个文件或者目录的链接。

5.3 Linux 的链接文件

从图 5.4 中可以看到 Linux 的文件系统中存在索引节点（innode），在 Linux 系统中可以基于该索引节点来创建符号链接文件。

5.3.1 链接文件基础

用户可以使用 ln 命令来创建文件的链接，该操作实际上是给 Linux 系统中已有的某个文件指定另外一个可用于访问它的名称，同时可以为这个新的文件指定不同的访问权限，以控制对信息的共享和安全性的问题。如果链接指向目录，用户就可以利用该链接直接进入被链接的目录，而不用输入一大堆的路径名，并且即使删除这个链接也不会破坏原来的目录。

Linux 系统下的链接可以分为硬链接（Hard Link）和软链接（符号链接，Symbolic Link）两种。硬链接要求链接文件和被链接文件必须位于同一个文件系统中，并且不能建立指向目录的硬链接。


注 意

ln 命令默认建立硬链接，如果给 ln 命令加上“-s”选项，则建立符号链接。

硬链接只能引用同一文件系统中的文件。它引用的是文件在文件系统中的物理索引。当移动或删除原始文件时，硬链接不会被破坏，因为它所引用的是文件的物理数据，而不是文件在文件结构中的位置（删除链接不会删除源文件，删除源文件不会删除链接）。

符号链接是一个指针，指向文件在文件系统中的位置。符号链接可以跨文件系统，甚至可以指向远程文件系统中的文件。符号链接只是指明了原始文件的位置，用户需要对原始文件的位置有

访问权限才可以使用链接。如果原始文件被删除，所有指向它的符号链接也就都被破坏了。它们会指向文件系统中并不存在的一个位置（删除链接并不会删除源文件，删除源文件会删除链接）。

在 Linux 文件系统中，物理索引值相同的文件是硬链接文件，也就是说，对于不同的文件名，物理索引可能是相同的，而一个物理索引值可以对应多个文件。

5.3.2 硬链接操作函数

Linux 提供了 link 函数和 unlink 函数用于对硬链接进行操作，前者用于建立一个硬链接，后者用于删除一个已经存在的硬链接，对两个函数的标准调用格式说明如下，如果调用成功则返回 0，否则返回-1。

```
#include <unistd.h>
int link(const char *existingpath,const char *newpath);
int unlink(const char *pathname);
```

对函数的参数说明如下。

- existingpath：已经存在的文件名称。
- newpath：待建立的硬链接文件名称，这是一个新的文件名称，如果该链接文件已经存在，则会返回错误信息-1。
- pathname：待删除的硬链接文件名称。

5.3.3 符号链接操作函数

Linux 提供了 symlink 函数用于创建一个符号链接文件，对其标准调用格式说明如下，如果操作成功则返回 0，如果操作失败则返回-1。

```
#include <unistd.h>
int symlink(const char *oldpath, const char *newpath);
```

对函数的参数说明如下。

- oldpath：已经存在的文件。
- newpath：oldpath 指向的文件。

【例 5.9】使用 symlink 函数建立符号链接文件

例 5.9 是一个使用 symlink 函数建立符号链接文件的实例，符号链接文件和指向的目标文件由 argv 参数给出，通过对 symlink 函数返回值 ret 的判断来确定建立符号连接文件是否成功，其流程如图 5.13 所示。

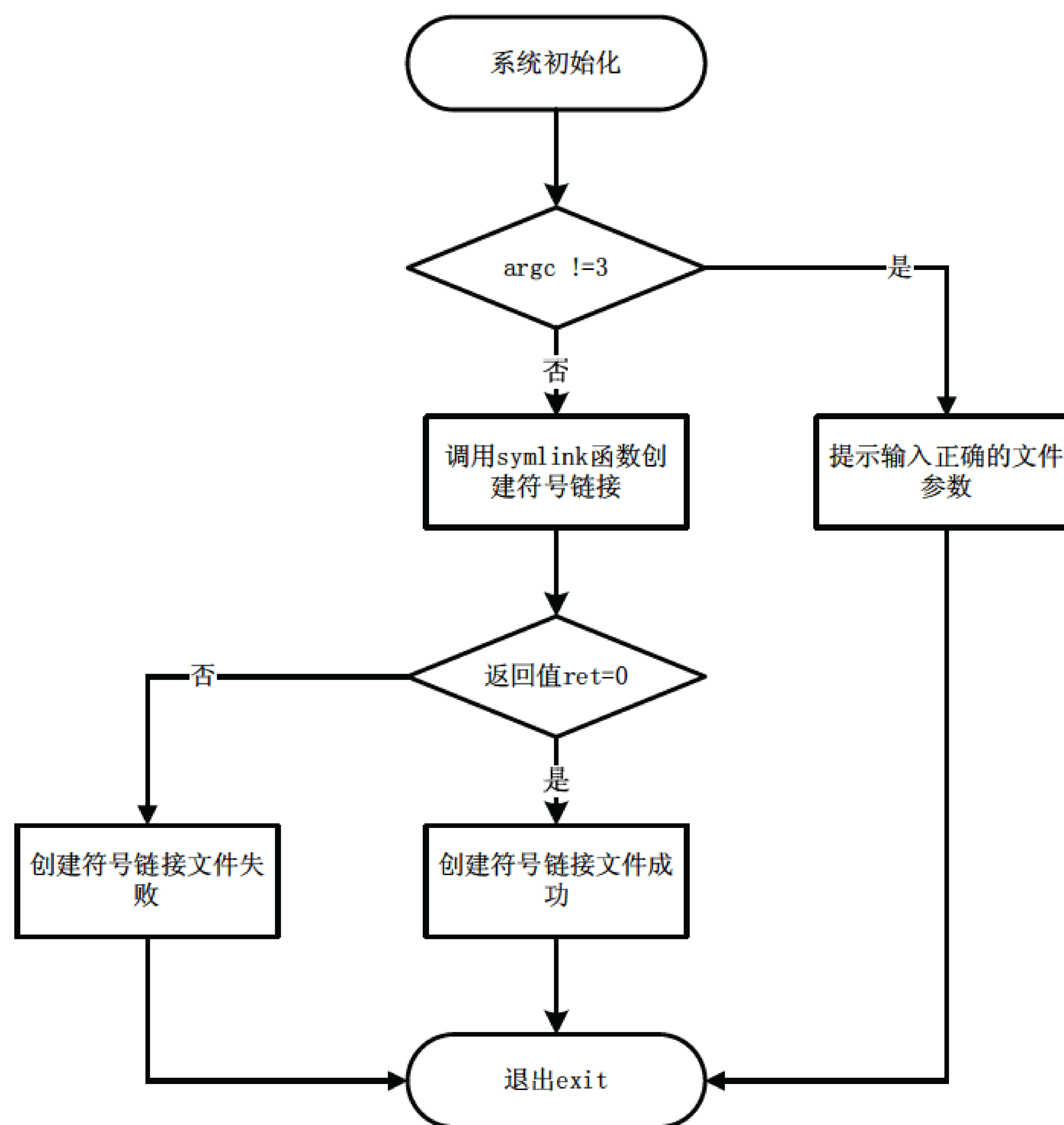


图 5.13　使用 symlink 函数建立符号链接文件

实例的应用代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc,char *argv[])
{
   int ret;                                //存放返回值
   if(argc != 3)                           //判断参数数目
   {
      printf("请输入 3 个参数\n");
      exit(0);
   }
   ret = symlink(argv[1],argv[2]);         //创建符号链接
   if(ret != 0)                            //创建符号链接文件失败
   {
      perror("创建符号链接失败!\n");
      exit(0);
   }
   return(0);
}
```

将文件保存为 exam509symlink.c，在终端中进行编译连接，生成可执行文件 exam509symlink。

```
alloy@ubuntu:~/linuxc/chapter5$ gcc exam509symlink.c -o exam509symlink
```

调用 exam509symlink 对例 5.1 中生成的可执行文件 exam501stat 生成一个符号链接文件 symlinkstat，然后调用“ls –l”命令查看 symlinkstat 文件的属性，可以看到 symlinkstat 文件已经链接到了 exam501stat 文件。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam509symlink exam501stat symlinkstat
alloy@ubuntu:~/linuxc/chapter5$ ls -l symlinkstat
lrwxrwxrwx 1 alloy alloy 11   2 月  21 03:48 symlinkstat -> exam501stat
```

由于已经建立了链接，所以 symlinkstat 文件其实质就是 exam501stat，这是一个对指定文件的类型进行判断的可执行文件，分别调用 exam501stat 文件和 symlinkstat 文件对例 5.3 中建立的 utimetest.txt 判断文件类型，可以看到如下的输出。

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam501stat utimetest.txt
这是一个普通文件！
alloy@ubuntu:~/linuxc/chapter5$ ./symlinkstat utimetest.txt
这是一个普通文件！
```

5.4 文件的其他操作

除了以上给出的相应函数外，Linux 还有一些其他的文件操作函数。

5.4.1 dup 和 dup2 函数

每一个打开的文件所对应的文件描述符都不是唯一的，同一个文件可能对应着多个文件描述符，而 dup 函数和 dup2 函数都可以用来复制文件描述符。

对 dup 和 dup2 函数的标准调用格式说明如下，如果操作成功则返回新的文件描述符，否则返回-1。

```
#include <unistd.h>
int dup (int fd);
int dup2 (int fd, int fd2);
```

由 dup 返回的新文件描述符一定是当前可用文件描述符中的最小数值。dup2 则可以用 fd2 参数指定新描述符的数值。如果 fd2 已经打开，则先将其关闭。若 fd 等于 fd2，则 dup2 返回 fd2，而不关闭它。通常使用这两个系统调用来重定向一个已打开的文件描述符。

【例 5.10】使用 dup 函数复制文件描述符

例 5.10 是一个使用 dup 函数复制文件描述的实例，应用代码首先调用 open 函数创建并且打开一个由 agrv 参数指定的文件，打印输出当前的文件描述符，然后调用 dup 函数获得新的文件描述符，如果调用成功，则打印输出新的文件描述符，其流程如图 5.14 所示。

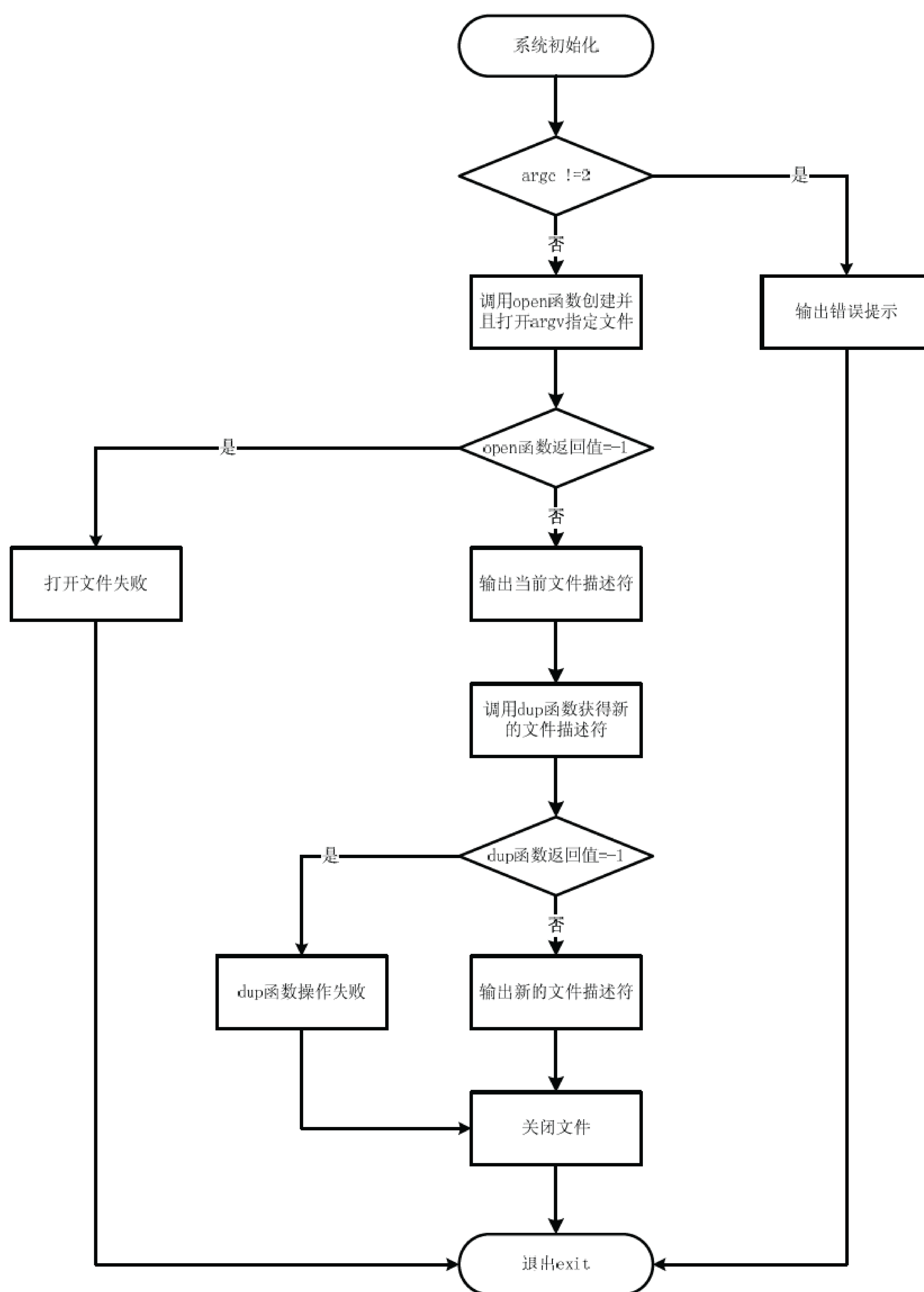


图 5.14　使用 dup 函数复制文件描述符

实例的应用代码如下：

```
1  //打开或者创建一个由 agrv 指定的文件
2  //然后使用 dup 函数复制该文件的描述符
3  //分别打印之前和复制之后的描述符
4  #include <stdio.h>
5  #include <unistd.h>
6  #include <fcntl.h>
7  #include <sys/stat.h>
8  #include <sys/types.h>
9  int main(int argc, char * argv[])
```

```
10  {
11      int fd;
12      if(argc != 2)     //如果参数错误
13      {
14          printf("参数错误\n");
15          return 1;
16      }
17      if((fd = open(*(argv+1),O_WRONLY|O_CREATE,0644)) == -1)   //打开 argv1 指定的文件
18      {
19          printf("打开文件%s 失败\n",*(argv+1));              //打开函数错误
20          return 2;
21      }
22      printf("当前文件描述符是%d\n",fd);                      //打印当前的文件描述符 w
23      if( (fd = dup(fd)) == -1)                               //获得新的文件描述符
24      {
25          printf("dup 文件错误\n");                           //dup 函数操作错误
26          return 3;
27      }
28      printf("dup 文件成功!\n");                              //dup 操作成功
29      printf("新的文件描述符是%d\n",fd);                      //打印新的文件描述符
30      close(fd);                                              //关闭文件
31      return 0;
32  }
```

将文件保存为 exam510dup.c，在终端中调用 gcc 进行编译链接，生成 exam510dup 可执行文件。

```
alloy@ubuntu:~/linuxc/chapter5$ gcc exam510dup.c -o exam510dup
```

调用 exam510dup 新建一个名为 duptest 的文件，并且使用 dup 函数获取其文件描述符，执行过程如下：

```
alloy@ubuntu:~/linuxc/chapter5$ ./exam510dup duptest
当前文件描述符是 3
dup 文件成功!
新的文件描述符是 4
```

5.4.2 fcntl 函数

fcntl 函数提供了进一步管理低级文件描述符的各种手段，用它可以对已打开的文件描述符执行各种控制操作，包括修改打开文件的性质、复制文件描述符、操作文件锁等。对 fcntl 函数的标准调用格式说明如下，如果调用成功，则其返回值根据 cmd 参数来决定，如果调用失败则返回-1。

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, int arg)
```

fcntl 函数的 fd 参数是打开的需要进行操作文件的文件描述符，而 cmd 参数决定了 fcntl 的功能和返回值，对其说明如表 5.9 所示。

表 5.9 fcntl 函数的 cmd 参数说明

选项	说明
F_DUPD	返回大于或等于 arg 的最低序号的文件描述符。该功能可以由 dup 函数实现。新的文件描述符与旧的文件描述符可以互换使用。若调用成功，则返回值为新的文件描述符
F_GETFD	获得 close-on-exec 标志，如果最后一位是 0，则该标志没有设置，返回值为 0 或 1
F_SETFD	设置 close-on-exec 标志为指定的值 arg（只有最后一位有效，为 0 或 1）
F_GETFL	获得文件打开的方式，返回所有的标志位
F_SETFL	设置文件打开方式的标志，设置文件打开方式为参数arg指定的方式，仅能设置O_APPEND 和 O_NONBLOCK（或 O_NDELAY），有的系统还可以设置 O_SYNC，该标志被文件描述符所有的拷贝（由 dup 或 fcntl 函数的 F_DUPFD 产生）所共享
F_GETLK	获得本进程得到锁的第一个锁的 flock 结构
F_SETLK	获得离散的文件锁，不等待
F_SETLKW	获得离散的文件锁，必要时等待
F_GETOWN	返回当前接收 SIGIO 或 SIGURG 信号（signal）的进程 ID 或进程组，进程 ID 以负值返回
F_SETOWN	设置进程或进程组接收 SIGIO 和 SIGURG 信号，进程组 ID 以负值返回，进程 ID 用正值指定

fcntl 函数的第三个参数 arg 可能是一个整数，也可能是一个如下的结构体，其和 cmd 参数相关。

```
struct flock{
        long        l_start;        /* 块开始处的偏移量 starting offset */
        long        l_len;          /* 块长 */
        long        l_pid;          /* 锁的属主（进程）*/
        long        l_type;         /* 锁的类型：读/写等*/
        long        l_whence;       /* 块开始处的类型 */
};
```

在 cmd 取值为 F_GETFL 关键字的时候 fcntl 函数将返回文件状态标志，如表 5.10 所示。

表 5.10 fcntl 函数返回的文件状态标志

文件状态标识	说明
O_RDONLY	只读
O_WRONLY	只写
O_RDWR	读写
O_APPEND	每次写时追加
O_NONBLOCK	非阻塞模式
O_SYNC	等待数据和属性写完成
O_DSYNC	等待数据写完成
O_RSYNC	同步读写
O_FSYNC	等待写完成
O_ASYNC	异步 I/O 操作

【例 5.11】使用 fcntl 函数获取文件标志

例 5.11 是一个使用 fcntl 函数获取 argv 参数指定文件的描述符并且打印该文件的标志说明实例。应用代码使用 fcntl 获取了文件的描述符，然后使用 switch 语句对文件的类型进行判断，并且输出相应的判断结果。

实例的应用代码如下：

```
 1  //这是一个使用 fcntl 函数来对文件描述符进行操作的实例
 2  #include <stdio.h>
 3  #include <fcntl.h>
 4  #include <stdlib.h>
 5  int main(int argc, char *argv[])
 6  {
 7      int val;
 8      if (argc != 2)                                          //如果参数错误
 9      {
10         printf("请输入正确的参数!\n");
11      }
12      if ((val = fcntl((atoi(*argv+1)), F_GETFL, 0)) < 0)
13      {
14         printf("使用 fcntl 操作文件描述符错误%d", atoi(*(argv+1)));
15      }
16      switch (val & O_ACCMODE)                                //判断文件的类型
17      {
18         case O_RDONLY:
19             printf("只读\n");
20             break;
21         case O_WRONLY:
22             printf("只写\n");
23             break;
24         case O_RDWR:
25             printf("读写\n");
26             break;
27         default:
28             printf("未知的模式\n");
29      }
30      if (val & O_APPEND)
31      {
32         printf(",写时追加\n");
33      }
34      if (val & O_NONBLOCK)
35      {
36         printf(",非阻塞\n");
37      }
38  #if defined(O_SYNC)
39      if (val & O_SYNC)                                       //等待数据和属性写完成
40      {
41         printf(",同步写\n");
42      }
43  #endif
```

```
44  #if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC)
45      if (val & O_FSYNC)                                  //等待写完成
46      {
47          printf(", 等待写完成");
48      }
49  #endif
50      putchar('\n');
51      return 0;
52  }
```

将文件保存为 exam511fcntl.c，在终端中使用 gcc 编译链接，生成可执行文件 exam511fcntl，然后调用该可执行文件对例 5.3 中建立的 utimetest.txt 和例 5.9 中建立的 symlinkstat 文件进行测试。

```
alloy@ubuntu:~/linuxc/chapter5$ gcc exam511fcntl.c -o exam511fcntl
alloy@ubuntu:~/linuxc/chapter5$ ./exam511fcntl
请输入正确的参数!
读写

alloy@ubuntu:~/linuxc/chapter5$ ./exam511fcntl utimetest.txt
读写

alloy@ubuntu:~/linuxc/chapter5$ ./exam511fcntl symlinkstat
读写
```

5.4.3 truncate 和 ftruncate 函数

可以调用 truncate 或者 ftruncate 函数来对文件的长度进行修改，其会影响到 stat 结构体中的 st_size 分量，对 truncate 和 ftruncate 函数的标准调用格式说明如下，如果调用成功返回 0，否则返回-1。

```
#include <unistd.h>
int truncate(char *pathname, size_t len);
int ftruncate(int fd, size_t len);
```

其中的参数 len 用于指定要将文件截取到的长度，pathname 参数对应的是文件名路径，而 fd 参数对应的是文件描述符。

5.5 本章习题

1. 编写一个程序，使用 stat 命令来获得/etc/passwd 文件的类型并且在屏幕上输出其大小。
2. 编写一个程序，打开一个指定文件，将它截断至 0 长度，但维持它的访问时间和修改时间不变。
3. 编写一个程序，用于测试 umask 函数的返回值。
4. 编写一个程序，用于测试如果 rename 函数的 oldname 和 newname 参数使用同一个参数的时候 rename 的返回值。
5. 编写一个程序，使用 dup 函数复制一个已经用 open 函数打开的文件描述符，然后将其关闭。

第 6 章　Linux 的流

前面介绍的使用 open 等函数对文件进行操作的方式通常被称为不带缓冲的 I/O，这是因为每次调用相应的函数（read 或者 write 等）对文件进行操作时都会调用内核的系统调用，这样的操作由于每次都要通过内核对文件直接进行操作，所以操作效率较低。本章将介绍另外一种对文件的操作方式：流编程。首先对文件所映射的流进行操作，然后分阶段将相应的数据写入文件中，从而极大地提高相应的操作效率。本章涉及的内容包括：

- Linux 中流的结构以及和文件的关系。
- Linux 的标准流。
- Linux 下使用 C 语言对流进行操作的方法。

6.1　Linux 的流基础

前面介绍的对文件的操作都是基于文件描述符的，而 Linux 的流操作都是基于流（stream）进行的，当利用标准 I/O 库打开（创建）一个文件的时候，Linux 系统其实已经使得一个流和该文件关联起来了。Linux 提供了大量的流操作库函数以供读者使用，这些库函数又被称为标准 I/O 库，这是因为这些库函数是跨操作系统平台的，并且是属于 ISO C 的组成部分。

6.1.1　流和文件的关系

文件的 I/O 函数都是针对文件描述符进行操作，当调用 open 或者其他函数打开一个文件时，即返回一个文件描述符 fd，然后针对该文件描述符进行后续的 I/O 操作，但是由于其需要多次反复调用系统，故而效率很低，表 6.1 是在调用 read 函数时使用不同缓冲区长度（size_t nbytes）来读 103316352 字节文件所需要花费的时间。

表 6.1　不同缓冲区长度下的读文件效率

缓冲区长度（字节）	用户 CPU 时间（秒）	系统 CPU 时间（秒）	时钟时间（秒）	循环次数
1	123.90	161.65	288.64	103316352
2	63.10	80.96	145.81	51658176
16	7.86	10.27	18.76	6457272
64	2.11	2.48	6.76	1614318
512	0.27	0.41	7.03	201789
1024	0.17	0.23	7.84	100894
4096	0.03	0.16	6.86	25223
8192	0.01	0.18	6.67	12611
65535	0.02	0.19	6.92	1576

从以上表格可以看到选择一个合适缓冲区的重要性，而流 I/O 函数的操作则是围绕流（stream）进行的，当使用流 I/O 库打开或创建一个文件时，可以使得一个流与一个文件相结合，接下来的操作过程就和基于文件描述符的 I/O 操作过程十分相似：对流进行读、写、定位操作等，最后关闭流。

图 6.1 是流、文件、基于流的 I/O 操作和基于文件的 I/O 操作的关系。

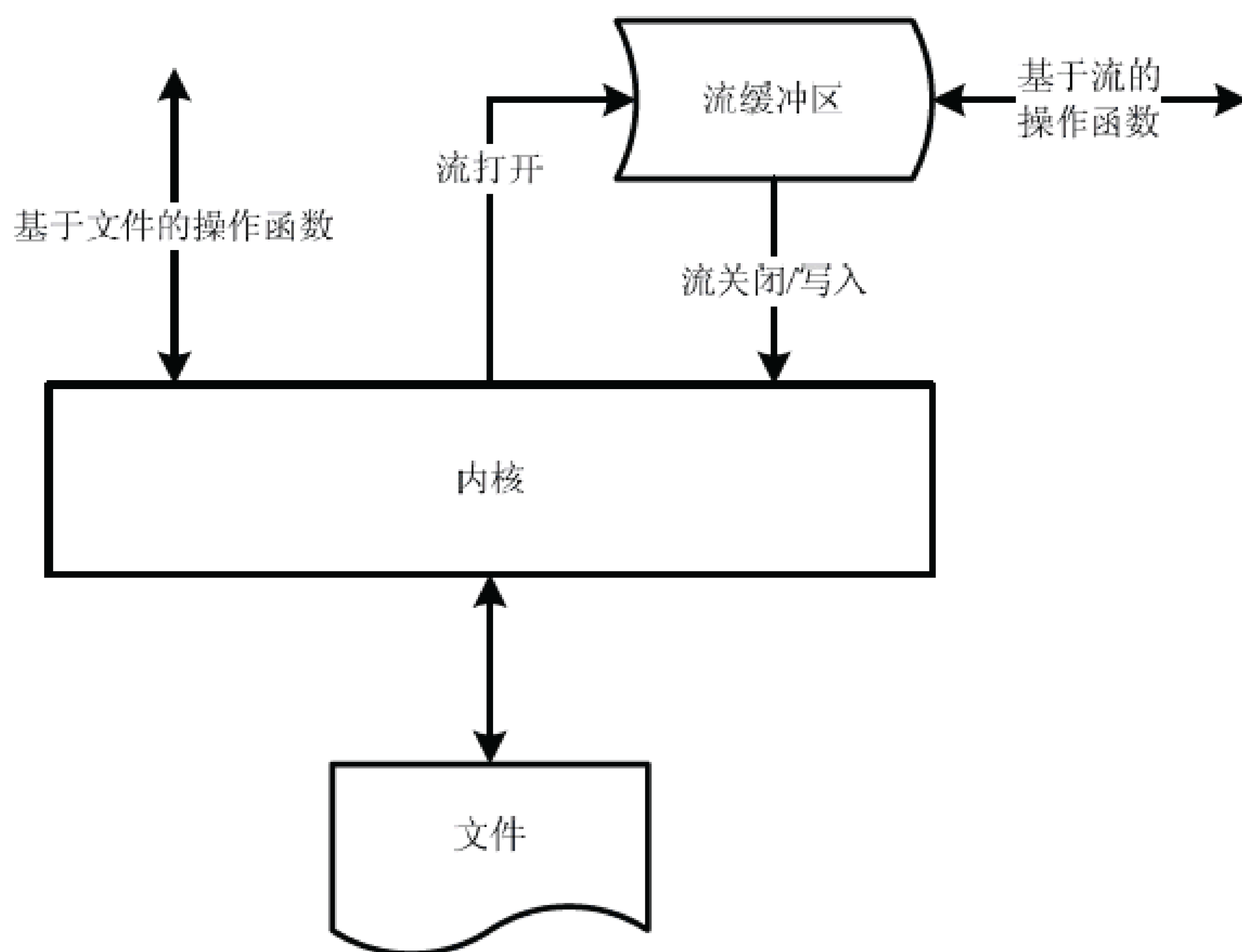


图 6.1　文件和流的关系

从图 6.1 中可以看到，所谓的带缓冲和不带缓冲是相对而言的。

不带缓冲的文件 I/O 操作也不是直接对文件进行的，只不过在用户层没有缓存区，所以被称为不带缓冲的 I/O，但对于 Linux 内核来说，还是进行了缓冲。当用户调用不带缓冲的 I/O 函数写入数据到文件的时候（即对磁盘存储区进行读写），Linux 内核会先将数据写入到内核中所设的缓冲存储器，假如该缓冲存储器的长度是 50 个字节，调用 write 函数进行写操作时，如果每次写入长度为 10 个字节，则需要调用 5 次 write 函数，而在这个过程中数据还是在内核的缓冲区中，并没有写入到磁盘，当 50 个字节已经写满了的时候才进行实际的 I/O 操作，即把数据写入到磁盘上。

带缓冲的 I/O 是在用户层建立了另一个缓存区（即流缓冲区），假设流缓存的长度同样也是 50 字节，当调用对应的写入库函数时会将数据写入到这个流缓存区里面，然后一次性进入内核缓存区，此时再使用系统调用将数据写入到文件（实质上是磁盘空间）上，从而减少了系统调用。

总之，对带缓冲和不带缓冲的 I/O 函数的单向数据（只有写入没有读出）流向可以总结如下：

- 不带缓冲：数据→内核缓存区→磁盘。
- 带缓冲：数据→流缓存区→内核缓存区→磁盘。

6.1.2　流的结构和操作流程

从上一小节可以知道，流操作函数的对象不是文件描述符，而是一个流缓冲区。当打开一个流时，返回一个指向 FILE 对象的指针，该对象通常是一个结构体，它包含了为管理该流所需要的所有信息，包括用于实际 I/O 的文件描述符、指向流缓存的指针、缓存的长度、当前在缓存中的字符数、出错标志等，对该结构体的说明如下：

```
struct file {
```

```
            struct list_head          f_list;
            struct dentry              *f_dentry;
            struct vfsmount            *f_vfsmnt;
            struct file_operations    *f_op;
            atomic_t                   f_count;
            unsigned int               f_flags;
            mode_t                     f_mode;
            loff_t                     f_pos;
            unsigned long              f_reada, f_ramax, f_raend, f_ralen, f_rawin;
            struct fown_struct         f_owner;
            unsigned int               f_uid, f_gid;
            int                        f_error;
            unsigned long              f_version;
            void                       *private_data;
            struct kiobuf              *f_iobuf;
            long                       f_iobuf_lock;
    };
```

用户的应用代码没有必要对 FILE 对象进行检验，在实际应用中也不需要了解 FILE 的结构，用户只需要知道为了引用一个流，应将 FILE 指针作为参数传递给对应的函数即可。

流操作的流程如图 6.2 所示，需要注意的是和基于文件的操作类似，在操作之后关闭流，否则容易导致数据的丢失。

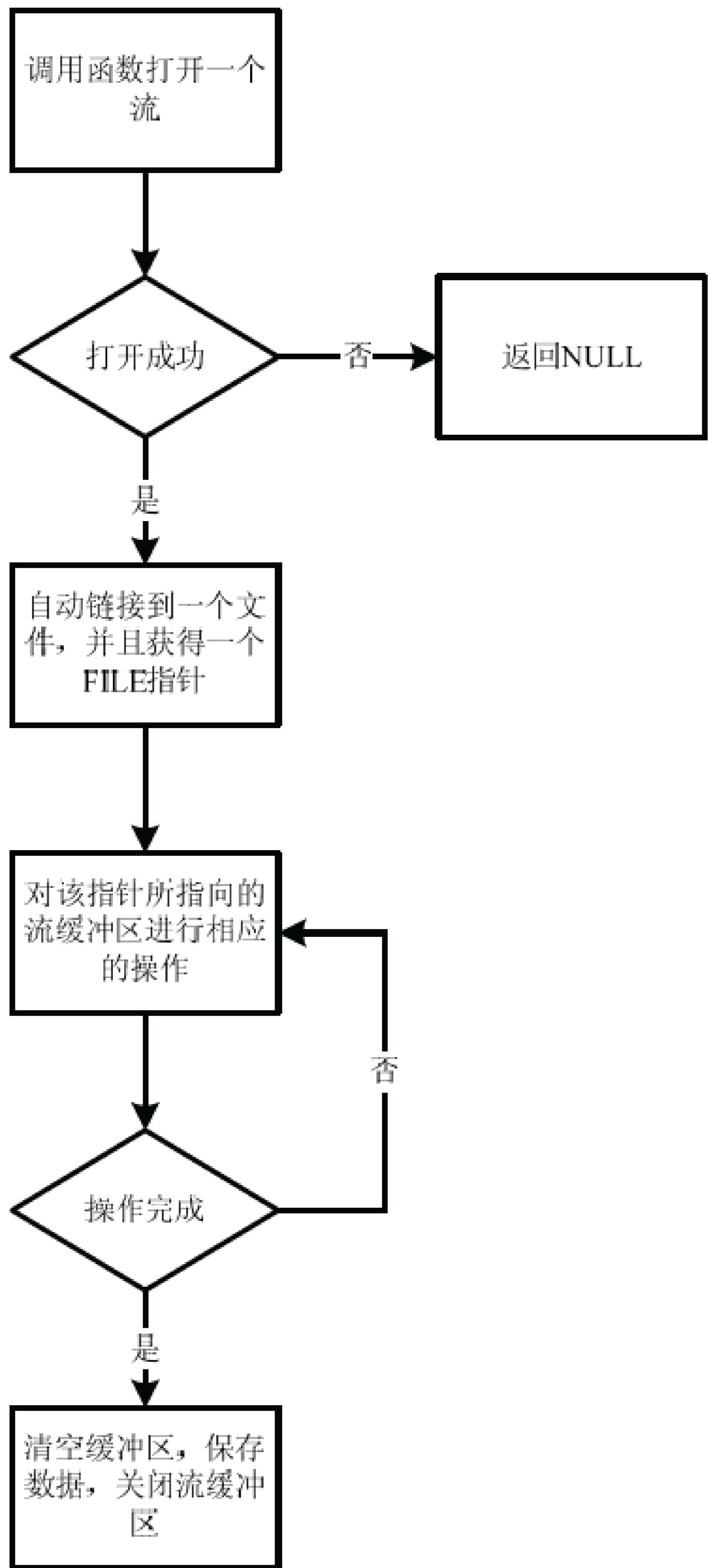


图 6.2　流操作流程


注 意

通常来说，用户可以简单地把流看做一块由系统分配的内存缓冲区，在该缓冲区中存放了文件对应的数据。

6.1.3 标准流介绍

在前面的内容中介绍了 Linux 有三个标准文件，分别为标准输入、标准输出和标准错误输出，Linux 操作系统对这三个标准文件预定义了三个标准流，可以通过相应的指针调用，对其说明如下：

```
#define    STDIN_FILENO     0    //标准输入，对应标准流指针 stdin
#define    STDOUT_FILENO    1    //标准输出，对应标准流指针 stdout
#define    STDERR_FILENO    2    //标准错误输出，对应标准流指针 stderr
```

需要注意的是，这三个标准流都是自动打开和自动关闭的。

6.2 流的基础操作

流的基础操作包括流的打开和关闭、管理流的缓冲方法、流的定位和读写等。

6.2.1 打开和关闭流

在对流进行操作之前，必须先打开这个流，而在操作完成之后，必须关闭流。

打开流的过程实际上是建立一个缓冲区，并且将这个缓冲区和对应的文件相关联的过程，Linux 提供了 fopen 系列函数来完成相应的工作，对其标准调用格式说明如下：

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
FILE *freopen(const char *path, const char *mode, FILE *stream);
```

对这三个函数的区别说明如下，当调用成功之后将返回一个 FILE 类型的文件指针，否则返回一个 NULL 指针。

- fopen 函数：打开一个指定的文件。
- freopen 函数：在一个指定的流上打开一个指定的文件，若该流已经打开，则先关闭该流，若该流已经定向，则立刻进行重定向操作。此函数一般用于将一个指定的文件打开为一个预定义的流：标准输入、标准输出或标准出错。
- fdopen 函数：打开一个由文件描述符所指定的流，此函数常用于由创建管道和网络通信通道函数获得的描述符，因为这些特殊类型的文件不能使用标准 I/O 的 fopen 函数打开，所以必须先调用设备专用函数以获得一个文件描述符，然后利用 fdopen 使一个标准 I/O 流与该描述符相结合。


注 意

通常来说 fopen 系列函数不会出错，其出错原因通常为：指定的文件路径有误，mode 参数非法以及对指定文件的操作权限不够。

流打开函数的参数说明如下。

- path 参数：文件的路径。
- fd 参数：文件描述符。
- stream 参数：指定的流。
- mode 参数：这个参数类似于 open 函数中的 mode 参数，用于说明流的打开模式和权限，对其详细说明如表 6.2 所示。

表 6.2 mode 参数说明

选项	说明
r 或 rb	只读打开，文件必须存在
w 或 wb	只写打开，如果该文件存在，则将它的长度截为 0，即该文件将被重新写过；如果该文件不存在，则创建一个新文件
a 或 ab	添加打开，若文件已经存在，则其原来的内容不变且到该流的输出将添加在文件的末尾，否则，创建一个新的空文件
r+、rb+或 r+b	读写打开，文件必须存在，该文件的原内容不变且初始文件位置位于文件开始之处
w+、wb+或 w+b	更新打开，若文件已存在，则其长度被截为 0，否则，创建一个新文件
a+、ab+或 a+b	更新打开，若文件已存在，则其原内容不变，否则，创建一个新文件，用于读的初始文件位置位于文件开始之处，但输出总是添加到文件的末尾

在表 6.2 中，使用关键标识符“b”来作为 mode 类型的参数用于区别二进制文件和文本文件，但是由于 Linux 内核并不对这两种类型的文件进行区分，所以其并没有实际的意义。另外对于 fdopen 函数来说，由于在获得描述符的时候该描述符已经被打开，所以如果使用“w”或者“wb”参数的时候并不截断该文件，另外使用“a”或者“ab”参数也不能用于创建一个文件，因为如果使用一个描述符来引用一个文件，则文件必须已经存在。

表 6.3 给出了打开一个流的 6 种不同方式，其中“×”代表不允许的操作，“●”表示允许的操作。

表 6.3 打开一个流的 6 种不同方式

限制条件	r	w	a	r+	w+	a+
文件必须已经存在	●	×	×	●	×	×
删除文件以前的内容	×	●	×	×	●	×
流可以读	●	×	×	●	●	●
流可以写	×	●	●	●	●	●
流只能在尾部写	×	×	●	×	×	●



注 意

在指定使用“w”或者“a”创建一个新文件的时候，并不能指定该文件的相应权限，如果需要对该文件进行相应的权限设置，必须调用 open 或者 create 函数。

当完成对一个流的操作之后，需要调用相应的函数将其关闭，Linux 提供了 fclose 函数用于该操作，对其标准调用格式说明如下：

```
#include <stdio.h>
int fclose(FILE *fp);
```

函数的参数是一个指向流的指针，调用成功之后返回“0”，否则返回“EOF”，其是一个定义为“-1”的宏。


注　意

EOF 也是 THE END OF THE FILE 的缩写，通常用来表示已经到达文件的结尾，将在后续章节中进行进一步介绍。

当调用 fclose 函数的时候，将会把流中的数据写入对应文件中，并且清除整个缓冲区。如果应用代码不调用该函数，则调用 exit 函数返回的时候，系统也会自动调用 fclose 函数完成对应的操作。

【例 6.1】打开和关闭指定流

例 6.1 是在 Linux 中调用 fopen 函数来创建一个由 agrv 参数指定的文件（流），然后关闭该文件（流）的实例，其流程如图 6.3 所示。

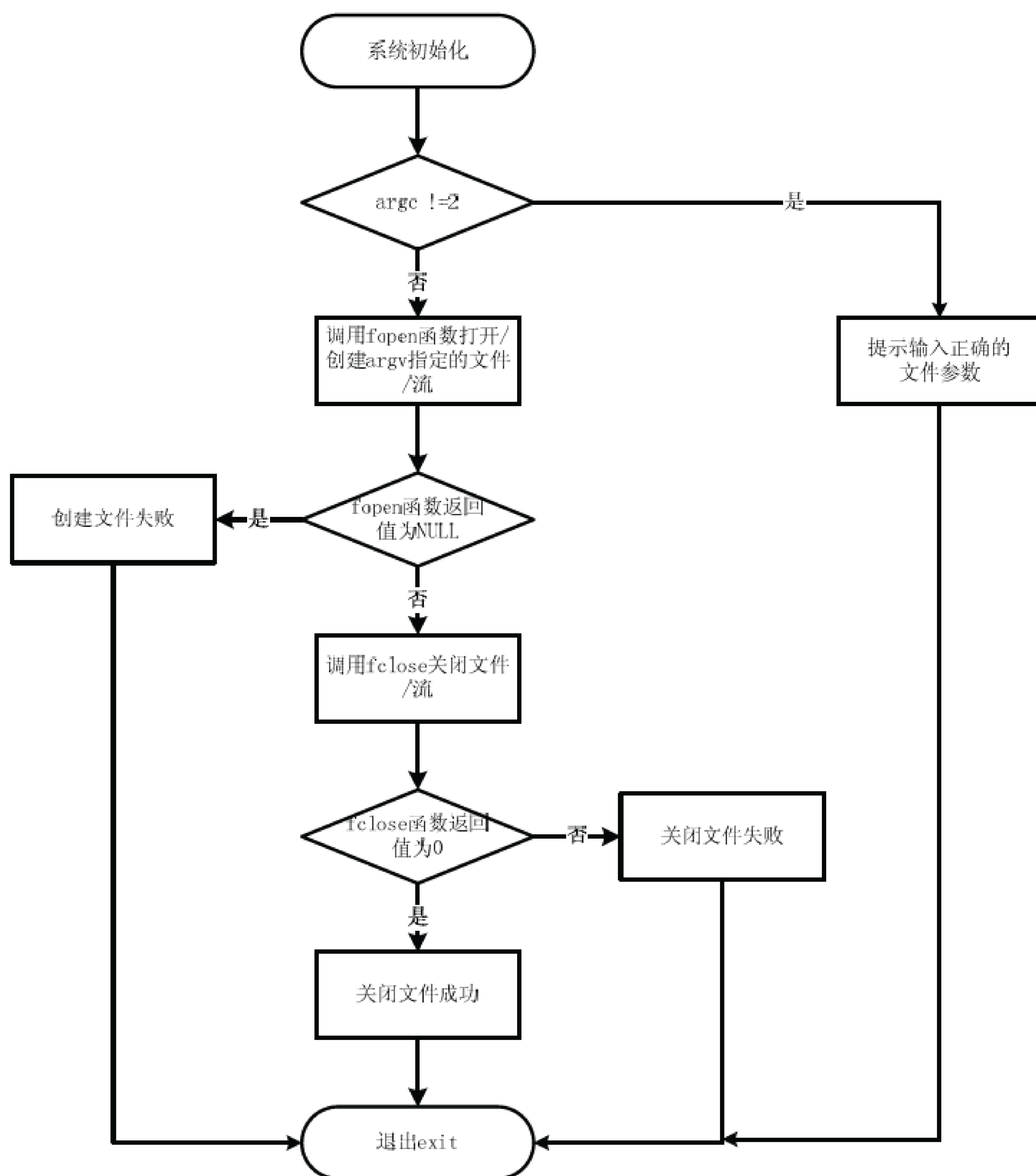


图 6.3　打开和关闭指定流

实例的应用代码如下：

```
1  //这是一个 fopen 和 fclose 函数的应用实例
2  //调用对应的流操作函数创建一个文件，并且关闭
3  //文件名由 argv[1]参数传递
4  #include <stdio.h>
5  int main(int argc, char *argv[])
6  {
7     FILE   *fp;                          //指向 FILE 对象的指针
8     int temp;                            //存放 fclose 函数的返回值
9     if(argc != 2)                        //如果参数不正确
10    {
11       printf("请输入正确的参数\n");
12       return 1;
13    }
14    fp = fopen( *(argv+1),"a+b");        //如果没有文件，则建立文件
15    if(fp == NULL)                       //如果 FILE 为 NULL，则表示失败
16    {
17       printf("创建文件%s 失败!", *(argv+1));
18       return 2;
19    }
20    printf("创建文件%s 成功!\n",*(argv+1));
21    temp = fclose(fp);                   //关闭文件
22    if(temp == 0)
23    {
24       printf("关闭文件%s 完成!\n",*(argv+1));
25       return 0;
26    }
27    else
28    {
29       printf("关闭文件%s 出错!", *(argv+1));
30       return 3;
31    }
32 }
```

将文件保存为 exam601fopen.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam601fopen。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam601fopen.c -o exam601fopen
```

在当前目录下使用 exam601fopen 创建一个名称为 fopentest 的文件，可以看到提示创建文件 fopentest 成功，使用“ls –l”查看该文件的属性。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam601fopen fopentest
创建文件 fopentest 成功!
关闭文件 fopentest 完成!
alloy@ubuntu:~/linuxc/chapter6$ ls -l fopentest
-rw-rw-r-- 1 alloy alloy 0   2 月  21 22:27 fopentest
```

既然 Linux 中的流最终都是需要对应到具体的文件（需要注意，标准输出设备、输入设备在 Linux 中也是以文件形式存在的），所以每一个流都有其对应的文件描述符，可以对流调用 fileno 函数来获得流对应的文件描述符，对其标准调用格式说明如下：

```
#include <stdio.h>
```

```
int fileno(FILE *stream);
```

函数的参数 stream 是一个流，返回值是该流对应文件的文件描述符。

【例 6.2】获得流对应文件的文件描述符

例 6.2 是使用 fileno 函数获得流对应的文件描述符的实例，应用代码首先使用 fopen 函数打开或者创建由 argv 参数指定的文件，然后使用 fileno 函数对 fopen 函数返回的流指针 fp 进行操作，以获得该文件对应的文件描述符，最后调用 fclose 函数来关闭文件，其流程如图 6.4 所示。

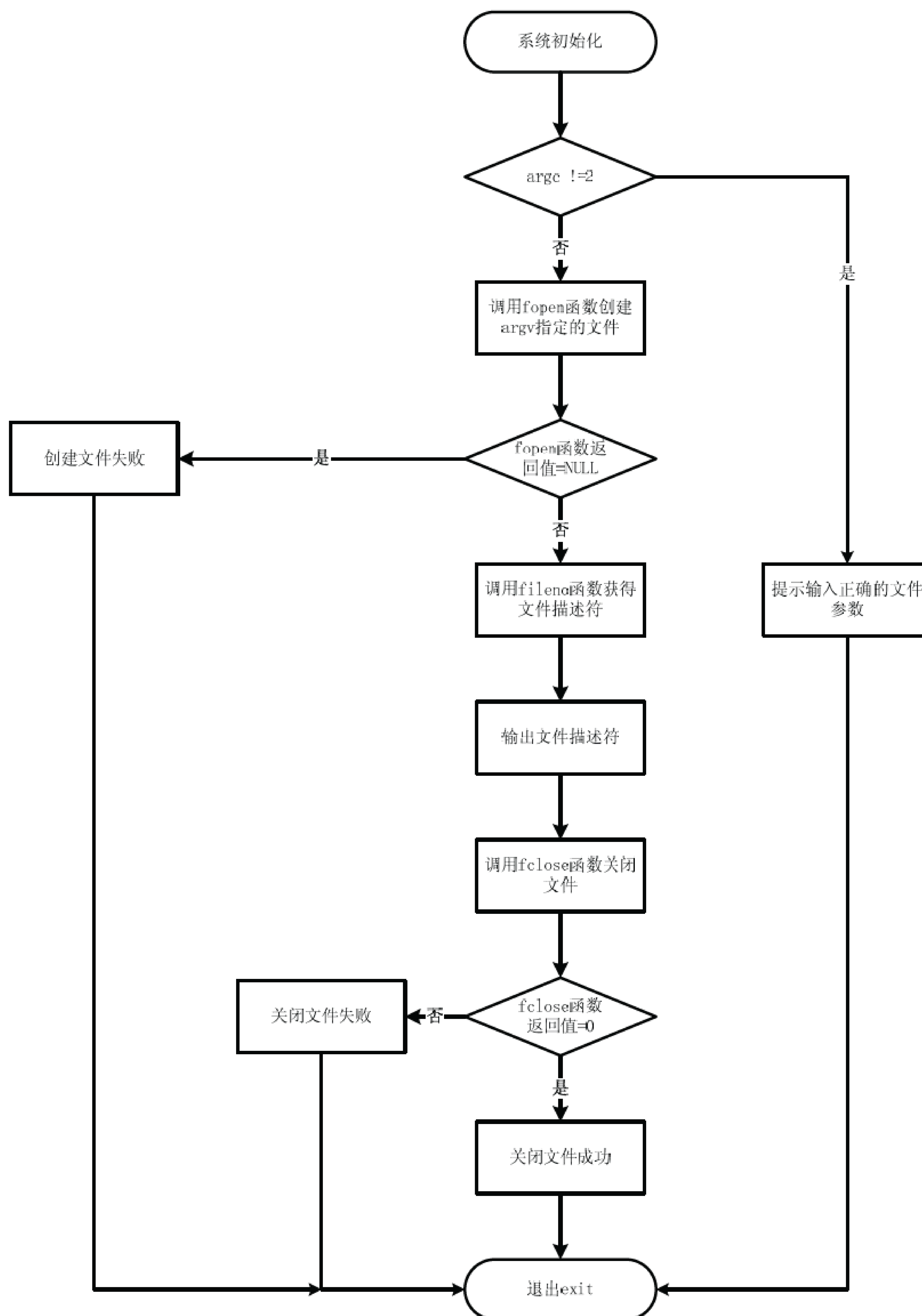


图 6.4　获取流对应文件的文件描述符

实例的应用代码如下：

```
1  //这是一个 fileno 函数的应用实例
2  //调用对应的流操作函数创建一个文件
3  //输出该流对应文件的描述符
4  //文件名由 argv[1]参数传递
5  #include <stdio.h>
6  int main(int argc, char *argv[])
7  {
8     FILE   *fp;                                   //指向 FILE 对象的指针
9     int temp;                                     //存放 fclose 函数的返回值
10    int fd;                                       //文件描述符
11    if(argc != 2)                                 //如果参数不正确
12    {
13       printf("请输入正确的参数\n");
14       return 1;
15    }
16    fp = fopen( *(argv+1),"a+b");                 //如果没有文件，则建立文件
17    if(fp == NULL)                                //如果 FILE 为 NULL 则表示失败
18    {
19       printf("创建文件%s 失败!", *(argv+1));
20       return 2;
21    }
22    else
23    {
24       printf("创建文件%s 成功!\n",*(argv+1));
25       fd = fileno(fp);                           //获得文件描述符
26       printf("文件%s 的文件描述符是%d\n",*(argv+1),fd);
27       temp = fclose(fp);                         //关闭文件
28       if(temp == 0)
29       {
30          printf("关闭文件%s 完成!\n",*(argv+1));
31          return 0;
32       }
33       else
34       {
35          printf("关闭文件%s 出错!", *(argv+1));
36          return 3;
37       }
38    }
39 }
```

将文件保存为 exam602fileno.c，在终端中使用 gcc 进行编译，生成可执行文件 exam602fileno。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam602fileno.c -o exam602fileno
```

调用可执行文件 exam602fileno，在当前目录下创建一个文件 filenotest，然后输出对应的文件描述符。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam602fileno filenotest
```

```
创建文件 filenotest 成功!
filenotest 的文件描述符是 3
关闭文件 filenotest 完成!
```

6.2.2　读写流

对流的操作的主要目的是对流所指定的文件进行操作，所以流的读写是流最重要也是最常见的操作，对流的读写操作可以按照操作的缓冲区大小分为三种。

- 字符读写：每次读写一个字符数据，如果流是带缓存的，则由流 I/O 函数处理所有缓存。
- 行读写：当遇到换行符的时候，则将流中换行符之前的内容送到缓冲区中，即每次读写一行。
- 块（结构）读写：以块（结构）为单位进行读写。

1. 按照字符读写流

字符读写方式每次从流中读出或者写入一个字符的数据，字符读可以调用 getc 系列函数进行读操作，对其标准调用格式说明如下：

```
#include <stdio.h>
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
```

函数如果调用成功，则返回即将读取的下一个字符，如果已经到达文件结尾或者出错，则返回 EOF，其参数是一个指向流的指针 stream。

在前两个函数中，参数 fp 表示所要读入字符的文件，它们的区别是 getc 可被实现为宏，而 fgetc 不能实现为宏。这意味着：

- getc 的参数不应当是具有副作用的表达式。
- 因为 fgetc 一定是个函数，所以可以得到其地址，这就允许将 fgetc 的地址作为一个参数传送给另一个函数。
- 调用 fgetc 所需时间很可能长于调用 getc，因为调用函数通常所需的时间长于调用宏，事实上在<stdio.h>头文件中，getc 便是以宏定义的形式实现的，其编码具有较高的工作效率。

第三个函数 getchar 只能用来从标准输入流中输入数据，其作用相当于调用以 stdin 为参数的 getc 函数，即 getc(stdin)。

另外，这三个函数以 unsigned char 类型转换为 int 的方式返回下一个字符，即使最高位为 1 也不会使返回值为负，这样就可以返回所有可能的字符值，再加上一个已发生错误或已到达文件尾端的指示值。在<stdio.h>中的常数 EOF 被要求是一个负值，其值经常是-1，这就意味着不能将这三个函数的返回值存放在一个字符变量中，以后还要将这些函数的返回值与常数 EOF 相比较。

对应按字符读，Linux 内核同样提供了按字符写函数，对其标准调用格式说明如下：

```
#include <stdio.h>
```

```
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

若函数调用成功，则返回输出字符 c，若出错则为 EOF，对其参数说明如下。

- c 参数：需要输出的字符。
- stream 参数：接收输出的流指针。

与输入函数一样，putchar(c) 等同于 putc(c, stdout)，putc 可被实现为宏，而 fputc 不能实现为宏。

【例 6.3】使用 getc 和 putc 读写流

例 6.3 是一个使用 getc 函数和 putc 函数按照字符对流进行读写的实例，其从标准输入（stdin）键盘读入用户的输入字符，然后送到标准输出（stdout）显示器显示，其流程如图 6.5 所示，其中涉及 ferror 错误处理函数的应用，可以参考第 6.2.3 小节。

实例的应用代码如下：

```
 1  //字符 I/O 函数 getc 和 putc 的应用实例
 2  //实例从标准输入键盘读入字符，然后送到标准输出显示器
 3  #include <stdio.h>
 4  int main(int argc,char *argv)
 5  {
 6      int temp;                                        //存放 I/O 函数的返回值
 7      printf("输入字符，输入 Ctrl+D 则停止\n");        //输出提示符
 8      while ((temp = getc(stdin)) != EOF)              //如果没有接收到 EOF
 9      {
10          if (putc(temp, stdout) == EOF)               //如果 putc 函数返回 EOF
11          {
12              printf("字符输出发生错误\n");
13              return 1;
14          }
15      }
16      if (ferror(stdin) != 0)                          //如果标准输入出现错误
17      {
18          printf("输入出现错误\n");
19          return 2;
20      }
21      return 0;
22  }
```

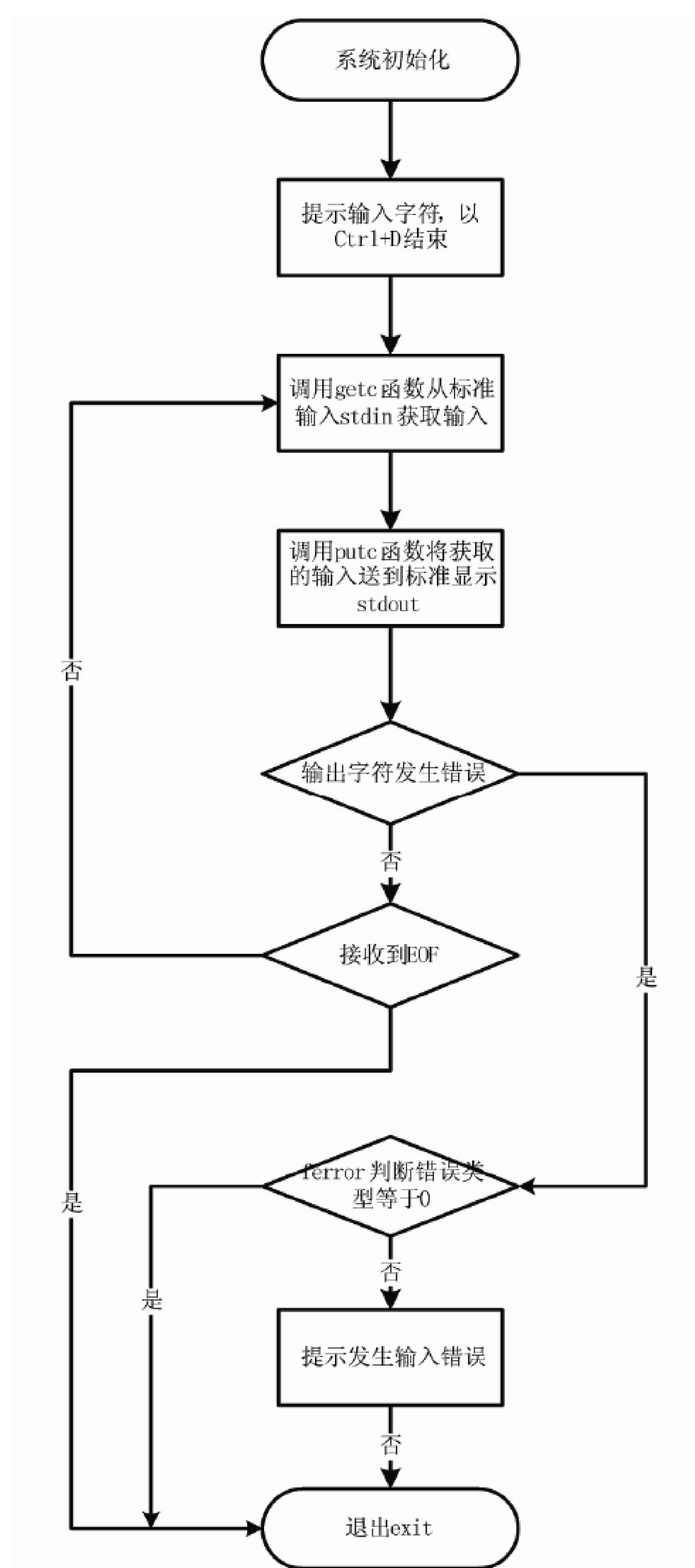


图 6.5　使用 getc 和 putc 函数对流进行读写

将文件保存为 exam603getcputc.c，在终端中使用 gcc 进行编译链接，生成 exam603getcputc 可执行文件。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam603getcputc.c -o exam603getcputc
```

在当前目录中直接执行 exam603getcputc 可执行文件，会看到提示输入字符，输入组合键 Ctrl+D 可以退出当前可执行文件。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam603getcputc
输入字符，输入 Ctrl+D 则停止
```

在终端中输入字符串后按回车键可以看到对应的字符串再次在显示终端中被输出，使用

Ctrl+D 组合键即可退出。

```
lslaerwoer
lslaerwoer
test
test
quit
quit
```

除了对标准输入输出流进行操作之外，还可以按照字符方式对一个文件进行操作，例 6.4 是调用 getc 和 putc 函数对一个指定文件进行操作的实例。

【例 6.4】使用 getc 和 putc 读写文件

本应用实例调用 fopen 函数打开一个由 argv 指定的文件，然后调用 getc 读出文件全部内容直到遇到 EOF 文件结尾，将读出的内容发送至屏幕显示，同时将存放在 writebuf 缓冲区中的字符串调用 putc 函数写入到文件中，最后关闭，其流程如图 6.6 所示。

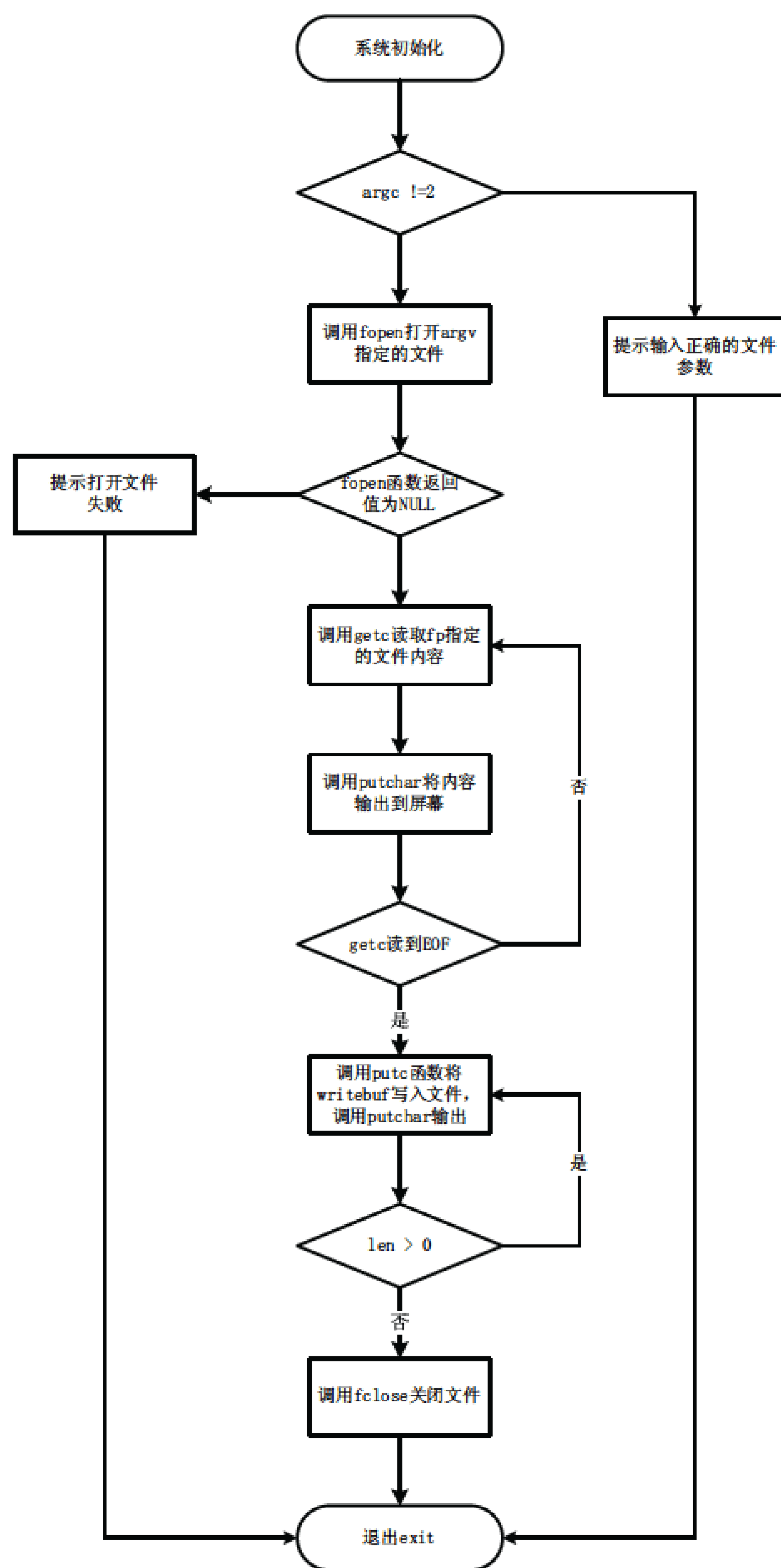


图 6.6　使用 getc 和 putc 读写文件

实例的应用代码如下：

```
1  //使用 fopen 打开指定文件
2  //调用 getc 读出数据并且显示到屏幕
3  //将一个字符串写入该文件
4  #include<stdio.h>
5  #include<string.h>
6  #include<stdlib.h>
7
8  int main(int argc,char *argv[])
9  {
10     int ch;
11     int len;                                      //写入缓冲区的长度计数器
12     int i = 0;
13     FILE *fp;                                     //文件结构指针
14
15     char writebuf[] = "Hello!I have read this file.\r\n";   //写入缓冲区
16     if(argc != 2)
17     {
18       printf("请输入正确的参数/n");               //参数错误
19       return 1;
20     }
21     fp = fopen(*(argv+1),"ab+");                  //打开指定文件
22     if(fp == NULL)
23     {
24         printf("打开文件%s 失败!\n",*(argv+1));
25         return 2;
26     }
27                                                 //从文件中读取数据，直到文件末尾
28     while( (ch = getc(fp)) != EOF)
29     {
30         putchar(ch);                            //在显示器上输出字符
31     }
32     //putchar('\n');                            //回车换行
33     len = strlen(writebuf);                     //获得写入缓冲区的实际长度
34     while(len > 0)                              //循环写入数据
35     {
36         putc(writebuf[i],fp);                   //写入数据
37         putchar(writebuf[i]);                   //显示数据
38         len--;
39         i++;                                    //更新计数器
40     }
41     fclose(fp);                                 //关闭计数器
42     return 0;
43 }
```

将文件保存为 exam604getcfile.c，在终端中使用 gcc 编译链接，生成可执行文件 exam604getcfile。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam604getcfile.c -o exam604getcfile
```

在当前工作目录下使用 vim 建立一个名为 getcputtest.txt 的文件，输入字符串“this is a test for char read stream”，然后保存，通过 cat 命令查看该文件内容以保证文件内容无误。

```
alloy@ubuntu:~/linuxc/chapter6$ vim getcputctest.txt
alloy@ubuntu:~/linuxc/chapter6$ cat getcputctest.txt
this is a test for char read stream
```

调用 exam604getcfile 文件对该.txt 文件进行读写操作，可以看到读出的字符串即为刚刚在 vim 中编辑的字符串，同时位于 writebuf 中的字符串“Hello!I have read this file.”也已经被写入了该文件。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam604getcfile getcputctest.txt
this is a test for char read stream
Hello!I have read this file.
```

再次使用 cat 命令查看该.txt 文件内容，可以看到 writebuf 中的字符串已经被写入。

```
alloy@ubuntu:~/linuxc/chapter6$ cat getcputctest.txt
this is a test for char read stream
Hello!I have read this file.
```

2. 按照行读写流

行读写方式每次从流中读出或者写入一行的数据，行读可以使用 gets 系列函数，对其标准调用格式说明如下：

```
#include <stdio.h>
char *fgets(char *s, int size, FILE *stream);
char *gets(char *s);
```

fgets 函数用于从流 stream 中读出一行数据，并且送到由 s 指定的缓冲区中，缓冲区大小由 size 参数说明，函数一直读到遇到下一个换行符或者读完了 n−1 个字符。如果需要读入行超过了 n−1 个字符，则只返回一个不完整的行，但是这个缓冲区总是以 NULL 结尾，下一次读取会继续执行，如果操作成功则返回缓冲区，如果已经到达文件结尾或者出错，则返回 NULL。

fget 函数和 fgets 函数功能类似，不过其是从标准输入流读取数据。


注 意

在实际使用中，并不推荐使用 fgets 函数，这是因为该函数不能指定缓冲区的大小，在实际使用中容易造成缓冲区溢出。

和行读入相对，Linux 内核也提供了相应的行写入 puts 系列函数，对其标准调用格式说明如下：

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

函数 fputs 用于将一个以 NULL 符为终止的字符串去掉 NULL 后写到指定的流，需要注意的是该函数并不要求每次输出一行，因为其并不要求在 NULL 符之前必须是换行符，如果成功则返回

一个非负值，如果出错则返回 EOF。

puts 函数先将一个以 NULL 符终止的字符串去掉 NULL 后写入到标准输出，然后再写一个换行符。


注 意

虽然 puts 并不像 gets 那样容易导致错误，但还是应该尽量避免使用这个函数，因为其涉及第二次写入一个换行符的问题。

例 6.5 是使用 fgets 和 fputs 命令重写例 6.3 的实例。

【例 6.5】使用 gets 和 puts 读写流

应用代码调用 fgets 函数从标准输入（stdin）键盘读入用户的输入字符，保存到缓冲区 buf 中，然后调用 fputs 函数送到标准输出（stdout）显示器显示，行缓冲区的大小由 MAXLINE（宏定义为 4096 个字符）决定，其中涉及 ferror 错误处理函数的应用，可以参考第 6.2.3 小节。

实例的应用代码如下：

```
 1  //使用 fgets 从标准输入读入一行数据
 2  //然后使用 fputs 发送至标准输出显示
 3  #include <stdio.h>
 4  #include <stdlib.h>
 5  #define MAXLINE 4096                              //定义一行的最大字符长度
 6  int main(int argc,char *argv[])
 7  {
 8     char buf[MAXLINE];        //缓冲区大小
 9     printf("输入字符,输入 Ctrl+D 则停止\n");         //输出提示符
10     while (fgets(buf, MAXLINE, stdin) != NULL)     //如果从标准输入读数据成功
11     {
12        if (fputs(buf, stdout) == EOF)              //如果从标准输出发生错误
13        {
14           printf("字符输出发生错误\n");
15           return 1;
16        }
17     }
18     if (ferror(stdin)!= 0)                         //如果从标准输入发生错误
19     {
20        printf("输入出现错误\n");
21     }
22     return 0;
23  }
```

将文件保存为 exam605fgetsfputs.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam605fgetsfputs。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam605fgetsfputs.c -o exam605fgetsfputs
```

在当前目录下运行 exa605fgetsfputs，可以看到如下的执行过程：

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam605fgetsfputs
输入字符,输入 Ctrl+D 则停止
this is a test!
this is a test!
345345345
345345345
```

例 6.6 是使用 gets 和 puts 函数重写例 6.4 的实例。

【例 6.6】使用 gets 和 puts 读写文件

本应用实例调用 fopen 函数打开一个由 argv 指定的文件，然后调用 gets 读出文件全部内容，直到到达 EOF 文件结尾，将读出的内容放入 buf 缓冲区中，同时发送至屏幕显示；再将存放在 writebuf 缓冲区中的字符串调用 puts 函数写入到文件中，最后关闭文件，buf 缓冲区的大小同样由宏定义 MAXLINE 设定为 4096 个字符。

实例的应用代码如下：

```
 1  #include<stdio.h>
 2  #include<string.h>
 3  #include<stdlib.h>
 4
 5  #define MAXLINE 4096                              //定义一行字符的最大长度
 6  int main(int argc,char *argv[])
 7  {
 8      char buf[MAXLINE];                            //读写缓冲区
 9      int len;                                      //写入缓冲区的长度计数器
10      int i = 0;
11      FILE *fp;                                     //文件结构指针
12
13      char writebuf[] = "Hello!I have read this file.\n";   //写入缓冲区
14      if(argc != 2)
15      {
16        printf("请输入正确的参数/n");                //参数错误
17        return 1;
18      }
19      fp = fopen(*(argv+1),"ab+");
20      if(fp == NULL)
21      {
22          printf("打开文件%s 失败!\n",*(argv+1));
23          return 2;
24      }
25      //从文件中读取数据，直到文件末尾
26      while((fgets(buf,MAXLINE,fp)) != NULL)        //如果没有到文件末尾
27      {
28          fputs(buf,stdout);                        //在标准输出中输出字符
29      }
30      fputs(writebuf,fp);                           //将写入缓冲区的数据写入文件
31      fclose(fp);                                   //关闭计数器
32      return 0;
33  }
```

将文件保存为 exam606fgetsfile.c，然后在终端中使用 gcc 对其进行编译链接，生成可执行文件 exam606fgetsfile。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam606fgetsfile.c -o exam606fgetsfile
```

使用 cat 命令查看例 6.4 操作后的 getcputctest.txt 文件内容，可以看到如下的输出，其中最后一个字符串是调用例 6.4 中生成的可执行文件后写入文件的。

```
alloy@ubuntu:~/linuxc/chapter6$ cat getcputctest.txt
this is a test for char read stream
Hello!I have read this file.
```

在当前目录下使用刚刚生成的可执行文件 exam606fgetsfile 对文本文件 getcputctest.txt 进行操作，可以看到如下的输出，此处只有一个“Hello!I have read this file”的原因是在例 6.6 中并没有将 writebuf 的内容输出到屏幕上，这个输出的字符串还是在例 6.4 的执行中写入到文件的。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam606fgetsfile getcputctest.txt
this is a test for char read stream
Hello!I have read this file.
```

再次使用 cat 命令查看该文本文件，可以看到如下的输出，第二行的字符串“Hello!I have read this file”是刚刚被可执行文件 exam606fgetsfile 写入的。

```
alloy@ubuntu:~/linuxc/chapter6$ cat getcputctest.txt
this is a test for char read stream
Hello!I have read this file.
Hello!I have read this file.
```

3. 按照块/结构读写流

在读写操作中，如果需要操作的区域多于一个字符乃至多于一行，使用字符读写和行读写同样比较麻烦，并且如果在一行数据中包括了 NULL 字符，也会导致行操作的中止，此时可以使用按块/结构（二进制）读写函数。

对 Linux 提供的块/结构读写函数 fread 和 fwrite 的标准调用格式说明如下：

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,FILE *stream);
```

fread 函数用于执行直接输出操作，其参数 ptr 是指向读取数据的缓冲区指针，size 是读取对象的大小，nmemb 表示欲读取的对象个数，fp 是指向要读取的流的 FILE 结构指针，其返回值为读的对象数，如果出错或到达文件尾端，则此数字可以少于 nmemb。在这种情况下，应调用 ferror 或 feof 以判断究竟是哪一种情况。

fwrite 函数用于执行直接输入操作，参数 ptr 是指向存放将要输入数据的缓冲区指针，size 是写入对象的大小，nmemb 表示欲写入的对象个数，fp 是指向要写入的流的 FILE 结构指针，其返回值为写的对象数，如果返回值少于所要求的 nmemb，则出错。

fread 和 fwrite 函数有如下两种常见的用法：

- 读或写一个二进制数组，例如将一个浮点型数组的第 2~5 个元素写至一个文件上，对其代码结构说明如下。

```
float data[10];
if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
printf("fwrite error!\n");
//其中，指定 size 为每个数组元素的长度，nmemb 为欲写的元素数
```

- 读或写一个结构，对其代码结构说明如下。

```
struct
{
short count;
long total;
char name[NAME_SIZE];
} item;
if (fwrite(&item, sizeof(item), 1, fp) != 1)
printf ("fwrite error!\n");
//其中，指定 size 为结构的长度，nmemb 为 1（要写的对象数）
```

将这两个例子结合起来就可读或写一个结构数组。为了做到这一点，size 应当是该结构的 sizeof，而 nmemb 应是该数组中的元素数。


注 意

fread 函数和 fwrite 函数的最大问题是其只能用于读取同一系统上已经写入的数据，这是因为在一个系统上写入的数据可能需要在另外一个系统上运行，从而因为结构体偏移量和存储方式等原因导致错误出现。

例 6.7 是一个使用 fread 和 fwrite 函数读写文件的应用实例。

【例 6.7】使用 fread 和 fwrite 函数读写文件

应用代码首先调用 fopen 函数以只读方式打开 argv+1 指定的文件作为源文件，然后调用 fopen 函数以只写方式打开 argv+2 指定的文件作为目标文件，此后使用 fread 函数从源文件中读出文件内容存放到 buf 缓冲区中，再将缓冲区内容使用 fwrite 函数写到目标文件中，其流程如图 6.7 所示。

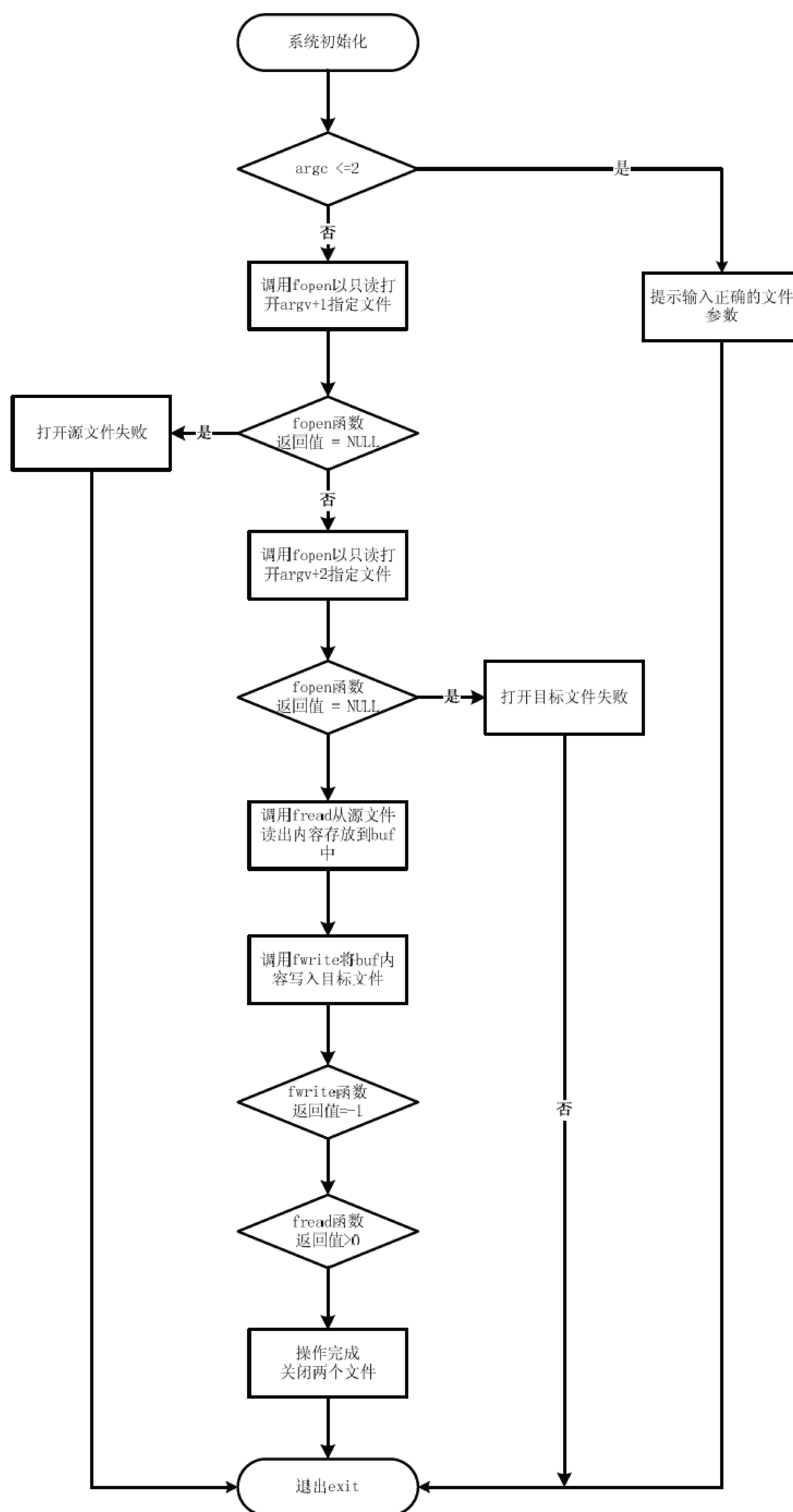


图 6.7 使用 fread 和 fwrite 函数读写文件

实例的应用代码如下：

```
1   #include   <stdio.h>
```

```
 2  #include    <stdlib.h>
 3  int main(int argc,char *argv[])
 4  {
 5       FILE *fp1, *fp2;                                 //流指针
 6       char buf[1024];                                  //缓冲区
 7       int n;                                           //存放 fread 和 fwrite 函数的返回值
 8       if(argc <=2)                                     //如果参数错误
 9       {
10           printf("请输入正确的参数\n!");                //参数错误
11       }
12       if ((fp1 = fopen(*(argv+1), "rb")) == NULL)
13       //以只读方式打开源文件，读的开始位置为文件开头
14       {
15             printf("读源文件%s 发生错误\n",*(argv+1));
16             return 1;                                  //出错退出
17         }
18       if ((fp2 = fopen(*(argv+2), "wb")) == NULL)
19       //以只写方式打开目标文件，写开始位置为文件结尾
20       {
21             printf("打开目标文件%s 失败\n",*(argv+2));
22             return 2;                                  //出错退出
23         }
24     //开始复制文件，文件可能很大，缓冲一次装不下，所以使用一个循环进行读写*/
25       while ((n = fread(buf, sizeof(char), 1024, fp1)) > 0)
26       {
27           //读源文件，直到将文件内容全部读完*/
28           if (fwrite(buf, sizeof(char), n, fp2) == -1)
29           {
30               //将读出的内容全部写到目标文件中去
31               printf("写入文件发生错误\n");
32               return 3;        /*出错退出*/
33           }
34       }
35       printf("从源文件%s 读数据写入目标文件%s 中完成\n",*(argv+1),*(argv+2));
                                                          //输出对应的提示
36       if(n == -1)
37       {
38             //如果因为读入字节小于 0 而跳出循环，则说明出错了*/
39             printf("读文件发生错误\n");
40             return 4;                                  /*出错退出*/
41       }
42       fclose(fp1);                                     /*操作完毕，关闭源文件和目标文件*/
43       fclose(fp2);
44       return 0;
45  }
```

将文件保存为 exam607freadfwritefile.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam607freadfwritefile。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam607freadfwritefile.c -o exam607freadfwritefile
```

调用 exam607freadfwritefile 对例 6.6 生成的文本文件 getcputctest.txt 进行操作，将其内容读出后写入一个新的文件 freadfwritetest.txt 中。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam607freadfwritefile getcputctest.txt freadfwritetest.txt
从源文件 getcputctest.txt 读数据写入目标文件 freadfwritetest.txt 中完成
```

操作完成之后调用 cat 命令分别查看这两个文本文件的内容，可以看到它们的内容完全相同。

```
alloy@ubuntu:~/linuxc/chapter6$ cat getcputctest.txt
this is a test for char read stream
Hello!I have read this file.
Hello!I have read this file.
alloy@ubuntu:~/linuxc/chapter6$ cat freadfwritetest.txt
this is a test for char read stream
Hello!I have read this file.
Hello!I have read this file.
```

6.2.3 流的出错处理

在第 6.2.2 小节中提到如果 fgets、gets、putc、fread 等函数失败会返回 EOF，但是由于 EOF 既用于报告文件结束，又用于报告随机出现的错误，因此，为了区分究竟是错误返回还是文件结束返回，有时还需要调用 ferror 函数来确定是否存在错误，调用 feof 函数检查是否遇到文件结束。

在大多数应用中，Linux 内核都为流（FILE）对象提供了两个标志符。

- 出错标志：当读写文件出错时该指示器被设置为真（非 0），否则为假（0）。
- 文件结束标志：当已经到达文件尾时该指示器被设置为真。

Linux 内核同样提供了 ferror 和 feof 函数用于检查这两个标志位，对其标准调用格式说明如下：

```
#include <stdio.h>
int feof(FILE *stream);
int ferror(FILE *stream);
```

feof 函数和 ferror 函数的参数都是一个指定的流指针，如果其测试标志位为真（非 0）则返回非 0 值，否则返回 0。

在确定了错误之后，可以调用 clearerr 函数来清除错误，对其标准调用格式说明如下，其参数是需要清除错误的流对应的指针，没有返回值：

```
#include <stdio.h>
void clearerr(FILE *stream);
```

例 6.8 是一个使用 feof 函数判断当前返回错误的实例。

【例 6.8】使用 feof 和 ferror 函数判断当前返回的错误

应用代码对一个空文件进行读操作，此时会返回一个 EOF 错误标志，调用 feof 函数和 ferror 函数判断到底是到了文件末尾还是出现错误，然后往文件里面写入一个字符串后再次判断，其流程

如图 6.8 所示。

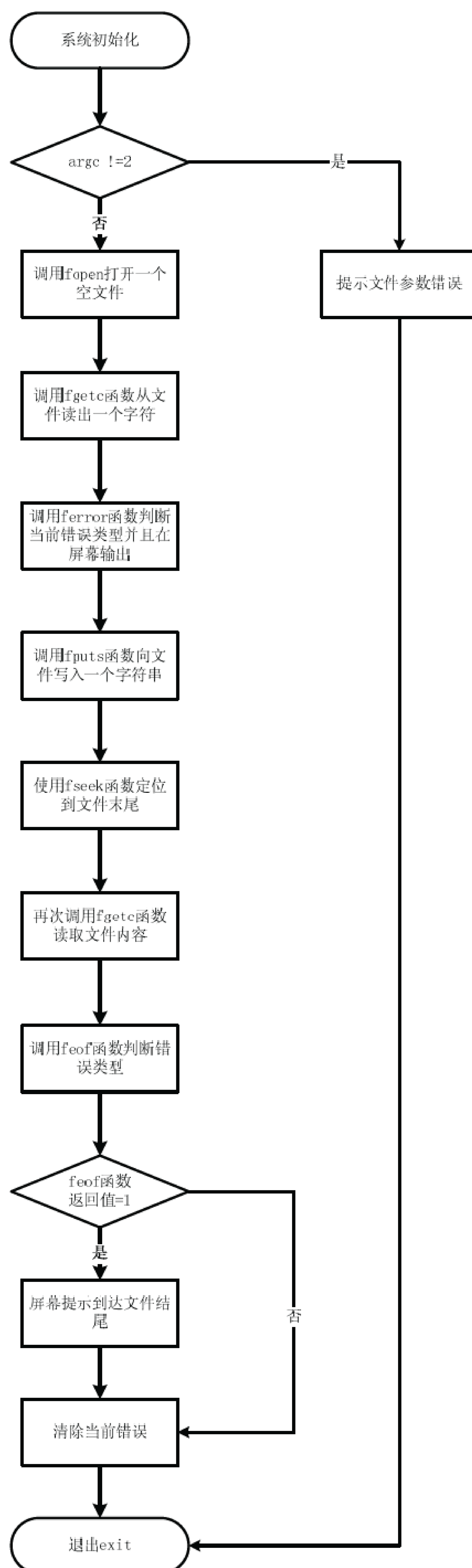


图 6.8　使用 feof 和 ferror 函数判断当前错误

实例的应用代码如下：

```
1   //判断到底是到了文件的结尾还是出错
2   #include <stdio.h>
3   #include <stdlib.h>
4   int main(int argc, char *argv[])
5   {
6      int i;
7      FILE *fp;
8      if(argc != 2)       //参数错误
9      {
10        printf("请输入正确的参数\n");
11        return 1;
12     }
13     fp = fopen(*(argv+1),"w+");
14     //打开文件，但是文件为空，所以无法读取，这时必须使用 w+作为参数，因为会截断文件
15     fgetc(fp);
16     //从文件中读出一个字符，文件为空，所以会报错
17     printf("ferror 的返回值为%d\n",ferror(fp));           //输出错误信息
18     fputs("abcdefgh",fp);                                 //向文件中写入一些数据
19     fclose(fp);                                           //关闭文件
20     fp=fopen(*(argv+1),"r");                              //再次打开文件
21     fseek(fp,0,SEEK_END);
22     //使用 fseek 定位到文件末位
23     fgetc(fp);                                            //读入
24     if(feof(fp) == 1)                                     //如果是到了末尾
25     {
26        printf("到达文件结尾\n");
27     }
28     clearerr(fp);                                         //清除当前错误
29     printf("ferror 的返回值为%d feof 的返回值为%d\n",ferror(fp),feof(fp));   //再次打印错误信息
30     fclose(fp);                                           //关闭文件
31     return 0;
32  }
```

将文件保存为 exam608feof.c，在终端中调用 gcc 对其进行编译链接，生成可执行文件 exam608feof。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam608feof.c -o exam608feof
```

在当前目录中使用 touch 命令创建一个空文件 zoo，使用“ls-l”命令可以查看该文件的长度为 0。

```
alloy@ubuntu:~/linuxc/chapter6$ touch zoo
alloy@ubuntu:~/linuxc/chapter6$ ls -l zoo
-rw-rw-r-- 1 alloy alloy 0   2 月  24 09:57 zoo
```

调用 exam608feof 对 zoo 进行操作，可以看到对应的错误编号输出，第一次返回的是文件空的错误标志，所以 ferror 函数的返回值是 1；第二次由于文件中已经有了字符串，但是由于当前已经

定位到了文件末尾，此时返回的是到达文件结尾错误，所以 feof 函数的返回值是 0。

```
ferror 的返回值为 1
到达文件结尾
ferror 的返回值为 0 feof 的返回值为 0
```


注 意

由于在应用代码的第 13 行中使用 fopen 函数在打开/创建文件的时候选用的参数是“w+”，如果该文件不存在，则新建一个文件的时候会生成一个空文件（参考表 6.2 和表 6.3），所以实际上不需要事先调用 touch 命令创建一个新文件，直接使用命令行创建/打开一个文件即可。

以下是两次调用 exam608feof 可执行文件对一个指定文件进行操作的输出：第一次调用 exam608feof 对文件 too 进行操作，此时 too 文件不存在，所以会首先创建 too 文件，但是此时 too 是一个空文件，所以 ferror 函数的返回值为 1；当执行完成之后调用“ls-l”命令可以看到此时 too 文件已经非空。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam608feof too
ferror 的返回值为 1
到达文件结尾
ferror 的返回值为 0    feof 的返回值为 0
alloy@ubuntu:~/linuxc/chapter6$ ls -l too
-rw-rw-r-- 1 alloy alloy 8    2 月  24 10:16 too
```

第二次调用 exam608feof 对文件 too 进行操作，此时 too 已经存在且非空，但是由于 fopen 函数使用 w+参数，所以在打开文件 too 时已经将文件截断成一个空文件，所以 ferror 函数的返回值还是 1。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam608feof too
ferror 的返回值为 1
到达文件结尾
ferror 的返回值为 0    feof 的返回值为 0
```

6.2.4 流的定位

和第 3.2.4 小节中介绍的文件偏移量类似，流在操作中也存在偏移量的概念，Linux 内核提供了如下 3 种方式来对流进行定位操作。

- 使用 ftell 和 fseek 函数：其缺点是必须假设偏移量可以放到一个长整型中。
- 使用 ftello 和 fseeko 函数：其使用了 off_t 数据类型替代了长整型。
- 使用 fgetpos 和 fsetpos 函数：使用一个抽象数据类型 fpos_t 来记录文件的位置，该数据类型可以定义为一个文件位置所需要的长度。

1. 使用 ftell 和 fseek 函数进行定位操作

对函数 ftell 和 fseek 的标准调用格式说明如下：

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
```

fseek 函数用于修改 fp 所指的文件偏移量，其中参数 fp 是流结构指针；参数 whence 指明参数 offset 的偏移起点，其参考值和 lseek 函数相同，如表 6.4 所示。参数 offset 是流的偏移值，其可以是一个正值，也可以是一个负值。如果函数调用成功则返回“0”，否则返回一个非“0”值。

表 6.4　whence 参数取值选项

选项	说明
SEEK_SET	文件位置定位于文件开始+offset 之处
SEEK_CUR	文件位置定位于文件当前位置+offset 之处
SEEK_END	文件位置定位于文件尾+offset 之处

如果 fseek 函数调用成功，则清除流的文件结束标志位（参考第 5.4.5 小节），如果该流是输出流并且缓冲的数据还未写至相连的文件，则 fseek 将导致未写出的数据被写至文件，因此，对于以更新方式（“+”）打开的文件而言，调用 fseek 之后，在此文件上的下一个操作既可以是输入，也可以是输出。

fseek 函数允许设置文件位置超过文件的当前文件尾，如果之后在此新文件位置写入了数据，则后续从原文件后与新写入的数据之间的空隙中读出的字节将用 0 填充，直至此空隙写入实际的数据为止。

ftell 函数的参数是需要操作的流指针，如果其调用成功，则返回 fp 所指定流的当前文件位置，它是从文件开始的字节数，否则返回“-1”。

针对 ftell 函数和 fseek 函数，Linux 还提供了 rewind 函数用于将偏移量设定到流的起始部分，对其标准调用格式说明如下：

```
#include <stdio.h>
void rewind(FILE *stream);
```

rewind 函数的参数是需要操作的流对应的指针，没有返回值，其等价于 fseek(fp,0L,SEEK_SET)。

例 6.9 是一个使用 fseek 函数在指定文件中连续写入字符串的实例，该实例和例 3.11 具有完全相同的效果。

【例 6.9】使用 fseek 函数获取文件偏移量

应用代码首先调用 fopen 函数，使用参数“a+b”（流只能在文件尾部写方式）打开 agrv 指定的文件，然后连续 10 次在文件尾部写入 writebuf[17]内部的字符串，缓冲区的大小由 sizeof 函数返回，这个值与循环次数计数器 i 相乘得到最后的总偏移量，使用 fseek 函数来定位当前写入的偏移量，使用 SEEK_SET 作为 fseek 函数的定位操作，其流程可以参考第 3 章的图 3.11。

实例的应用代码如下：

```
1   //这是一个使用 fseek 函数定位流文件中的位置
```

```
 2  //然后将一个字符串连续写入文件的应用
 3   #include <stdio.h>
 4   int main(int argc,char *argv[])
 5   {
 6     int temp,seektemp,i,j;
 7     FILE *fp;                              //流结构文件指针
 8     char writebuf[17] = "this is a test!\n";  //写缓冲区
 9     if(argc!= 2)                           //如果参数错误
10     {
11       printf("请输入正确的参数!\n");
12       return 1;                            //如果参数不正确则退出
13     }
14     fp = fopen(*(argv+1),"a+b");           //打开文件
15     for(i=0;i<10;i++)
16     {
17       j = sizeof(writebuf) * (i+1);        //计算下一次的偏移量
18       fseek(fp,j,SEEK_SET);
19       temp = fputs(writebuf,fp);           //写入数据，没有进行出错处理
20     }
21     fclose(fp);
22     return 0;
23   }
```

将文件保存为 exam609fseek.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam609fseek。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam609fseek.c -o exam609fseek
```

调用 exam609fseek 将 writebuf 中的字符串写入文件 fseektest.txt 中，然后使用“cat –n”命令查看该文件的内容，可以看到该字符串被连续写入了 10 次。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam609fseek fseektest.txt
alloy@ubuntu:~/linuxc/chapter6$ cat fseektest.txt -n
 1  this is a test!
 2  this is a test!
 3  this is a test!
 4  this is a test!
 5  this is a test!
 6  this is a test!
 7  this is a test!
 8  this is a test!
 9  this is a test!
10  this is a test!
```



注 意

写入缓冲区 writebuf 也可以定义为一个变长数组，即定义为 writebuf[]="this is a test!"，这并不会影响 sizeof 函数的返回结果。

2. 使用 ftello 和 fseeko 函数进行定位操作

对于二进制文件而言，其文件偏移量可以简单地利用一个长整型数据来确定，但是在文本文件中其当前位置不一定能以简单的字节偏移量来度量，这是因为虽然 Linux 并不区分二进制文件和文本文件，但是在其他的操作系统中这两个文件的存放格式可能是不同的，此时可以使用一个 off_t 类型的数据类型，并且使用 ftello 和 fseeko 函数，这两个函数除了位移量的数据类型不同之外，其他方面和 fseek、ftell 函数完全相同，对其标准调用格式说明如下：

```
#include <stdio.h>
int fseeko(FILE *stream, off_t offset, int whence);
off_t ftello(FILE *stream);
```

3. 使用 fgetpos 和 fsetpos 函数进行定位操作

fgetpos 和 fsetpos 两个函数同样可以用于定位流的操作，fgetpos 可以得到读写指针的位置，而 fsetpos 可以定位读写指针的位置，对其标准调用格式说明如下：

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);
```

在这两个函数中，参数 fp 是流指针，pos 为指向 fpos_t 的指针，pos_t 是一个存放指针位置的记录类型，如果操作成功则返回“0”，如果出错则返回非“0”值。

这两个函数和 ftell、fseek 函数的主要区别在于其使用了 fpos_t 结构来存放偏移值，这是一个抽象的结构体，可以在多种不同的操作系统中进行具体定义。

在 Linux 中 fpos_t 结构的定义位于/usr/include 的_G_config.h 文件中，对其定义说明如下，后者为 64 位系统下的结构定义，前者为 32 位的结构定义，这两个结构均由 off_t_pos 和 mbstate_t_state 两个分量组成。

```
typedef struct
{
  __off_t __pos;
  __mbstate_t __state;
} _G_fpos_t;
typedef struct
{
  __off64_t __pos;
  __mbstate_t __state;
} _G_fpos64_t;
```

例 6.10 是在例 6.9 的基础上使用 fgetpos 函数获取当前文件偏移量位置的实例。

【例 6.10】使用 fgetpos 函数获取文件当前偏移量

应用代码在例 6.9 的基础上，在每次将 writebuf 缓冲区数据写入文件之后使用 fgetpos 函数来获得当前的偏移量，然后将这个偏移量输出。

实例的应用代码如下：

```
1   //这是一个利用 fgetpos 来将一个字符串写入文件的实例
2   //在每次写入文件成功之后将当前文件偏移量给出
3   #include <stdio.h>
4   #include <stdlib.h>
5   int main(int argc,char *argv[])
6   {
7      int temp,seektemp,i,j;
8      FILE *fp;                                    //流文件结构指针
9      fpos_t ps;                                   //当前偏移量指针
10     char writebuf[] = "this is a test!\n";       //写缓冲区
11     if(argc!= 2)                                 //如果参数不正确
12     {
13        printf("请输入正确的参数!\n");
14        return 1;                                 //如果参数不正确则退出
15     }
16     fp = fopen(*(argv+1),"a+b");                 //打开文件
17     for(i=0;i<10;i++)
18     {
19        j = sizeof(writebuf) * (i+1);             //计算下一次的偏移量
20        fseek(fp,j,SEEK_SET);
21        temp = fputs(writebuf,fp);                //写入数据
22        fgetpos(fp,&ps);                          //获得当前的偏移量
23        printf("当前文件偏移量 fpos 为%ld \n",ps);  //打印当前的偏移量输出
24     }
25     fclose(fp);
26     return 0;
27  }
```

将文件保存为 exam610fgetpos.c，然后在终端中使用 gcc 进行编译链接，输出得到可执行文件 exam610fgetpos。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam610fgetpos.c -o exam610fgetpos
```

在当前目录下使用 exam610fgetpos 对一个新文件 getposfile.txt 进行操作，可以看到如下的输出，每次文件偏移量都发生了相应的字符数变化。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam610fgetpos getposfile.txt
当前文件偏移量 fpos 为 16
当前文件偏移量 fpos 为 32
……//其间省略部分内容
当前文件偏移量 fpos 为 144
当前文件偏移量 fpos 为 160
```

6.2.5 流的缓冲管理

和基于文件的 I/O 方式比起来，基于流的 I/O 方式的最大特点就是其先对缓冲区进行操作，从而可以大大提高效率，但是当使用 fopen 系列函数打开一个流的时候并没有指定这个流对应的缓冲方式和缓冲区大小，因为这是由 Linux 内核来分配的。

1. 管理流的缓冲方式

缓冲方式是指流在什么时候使用内核的系统写入/读出调用来对文件进行操作，有三种类型的缓冲方式。

- 全缓冲：在这种缓冲方式下直到缓冲区被填满，才使用系统调用进行操作。对于读操作来说，直到读入的内容字节数等于缓冲区大小或者文件已经到达结尾，才进行实际的 I/O 操作，将外存文件内容读入缓冲区；对于写操作来说，直到缓冲区被填满，才进行实际的 I/O 操作，缓冲区内容写到外存文件中。磁盘文件通常是全缓冲的。在 Linux 内核中使用宏定义_IO_FULL_BUF 来表示全缓冲，通常来说，在一个流上进行第一次读写操作的时候，会调用 malloc 内存分配函数来为流分配一块内存作为缓冲区域。
- 行缓冲：在这种缓冲方式下如果遇到换行符，可使用系统调用进行操作。对于读操作来说，遇到换行符‘\n’才进行 I/O 操作，将所读内容读入缓冲区；对于写操作来说，遇到换行符‘\n’才进行 I/O 操作，将缓冲区内容写到外存中。由于缓冲区的大小是有限的，所以当缓冲区被填满时，即使没有遇到换行符‘\n’，也同样会进行实际的 I/O 操作；标准输入 stdin 和标准输出 stdout 默认都是行缓冲的。在使用行缓冲的时候有两个限制：第一，每一行对应的缓冲区长度是固定的（MAXLINE，在 Linux 中这个值通常被定义为 4096 个字节），如果这个缓冲区已经被写满，即使还没有遇到换行符，也会调用系统进行工作；第二，在 Linux 内核要求获得数据的时候，将立即完成一次数据的写入或者读出。
- 无缓冲：在这种缓冲方式下没有缓冲区，不进行缓冲，数据会立即读入或者输出到外存文件和设备上。标准出错 stderr 是无缓冲的，从而保证错误提示和输出能够及时反馈给用户，以供用户排除错误。

Linux 下的流缓冲具有以下两个特征：

- 当且仅当输入和输出不涉及交互式设备的时候，其才是全缓冲的，如果涉及了终端设备，大部分将是行缓冲。
- 标准出错绝对不是全缓冲的。

Linux 中的流最终都是需要对应到具体的文件的（需要注意标准输出设备、输入设备这些在 Linux 中也是以文件形式存在的），所以每一个流都有其对应的文件描述符，可以对流调用 fileno 函数来获得流对应的文件描述符（参考例 6.2）。

例 6.11 是一个查看三个标准流的缓冲方式实例。

【例 6.11】查看三个标准流的缓冲方式

应用代码通过调用函数 pr_stdio 来分别打印三个标准流的缓冲方式，函数 pr_stdio 的流程如图 6.9 所示，其通过使用 if 语句对流属性的比较来输出对应的流缓冲方式。

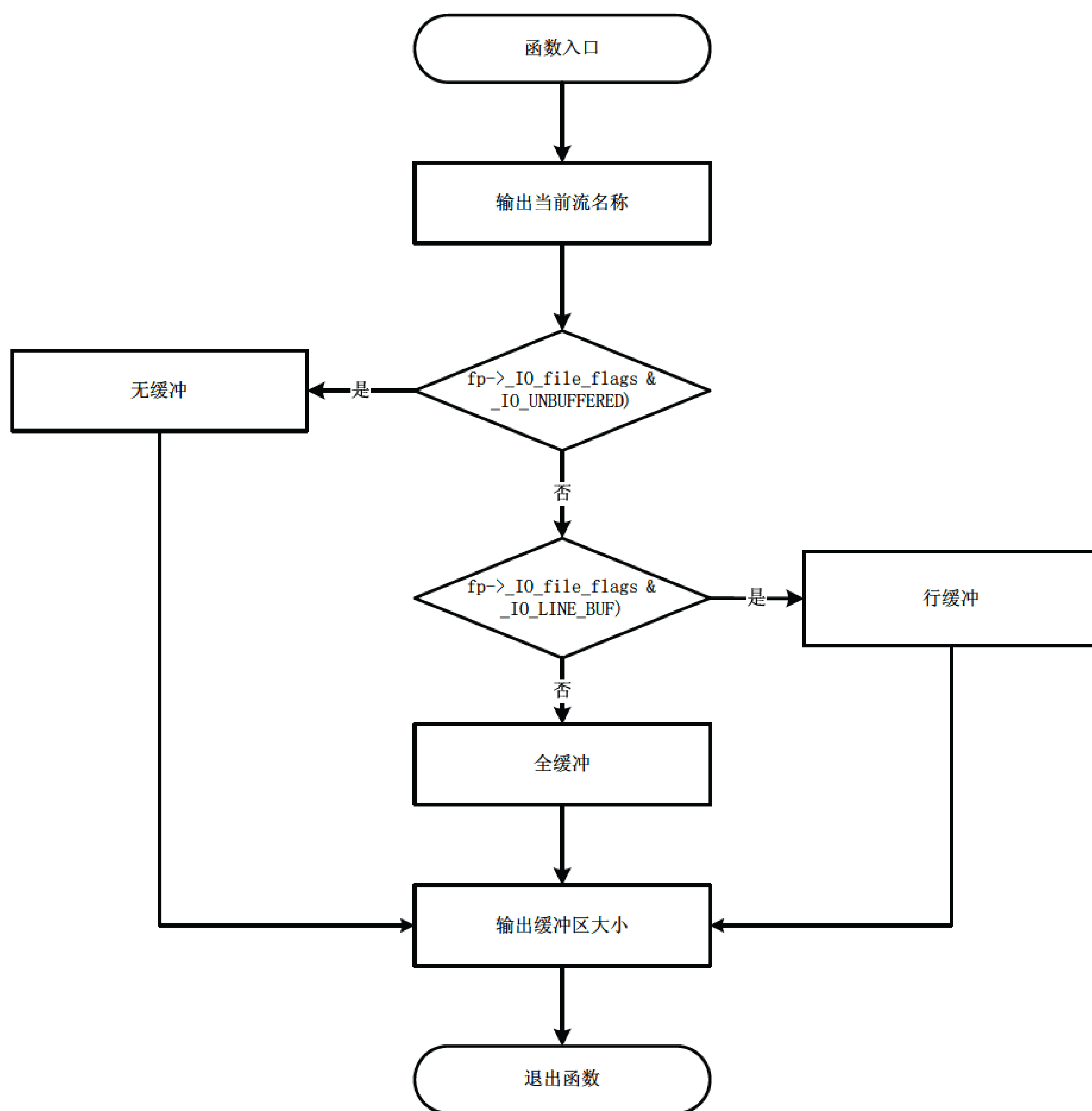


图 6.9　pr_stdio 函数的执行流程

实例的应用代码如下：

```
1   //这是一个分别打印三个标准流和一个文件流的缓冲方式的应用实例
2
3   #include <stdio.h>
4   #include <stdlib.h>
5
6   #if defined(MACOS)
7   #define _IO_UNBUFFERED      __SNBF
8   #define _IO_LINE_BUF        __SLBF
9   #define _IO_file_flags      _flags
10  #define BUFFERSZ(fp)        (fp)->_bf._size
11  #else
12  #define BUFFERSZ(fp)        ((fp)->_IO_buf_end - (fp)->_IO_buf_base)
13  #endif
14  //以上是关于缓冲方式和缓冲区大小的预定义
```

```
15  void pr_stdio(const char *, FILE *);
16  //子函数声明
17  int main(int argc,char *argv[])
18  {
19     FILE    *fp;                          //流文件结构指针
20     pr_stdio("stdin",    stdin);          //标准输入
21     pr_stdio("stdout", stdout);           //标准输出
22     pr_stdio("stderr", stderr);           //标准出错处理
23     printf("fopen error");
24     if (getc(fp) == EOF)
25     {
26        printf("getc error\n");
27     }
28     return 0;
29  }
30  //测试缓冲输出函数
31  void    pr_stdio(const char *name, FILE *fp)
32  {
33     printf("当前流是%s, ", name);          //打印流的名称
34     if (fp->_IO_file_flags & _IO_UNBUFFERED)
35     {
36        printf("无缓冲\n");
37     }
38     else if (fp->_IO_file_flags & _IO_LINE_BUF)
39     {
40        printf("行缓冲\n");
41     }
42     else
43     {
44        printf("全缓冲\n");
45     }
46     printf(", 缓冲区大小 = %ld\n", BUFFERSZ(fp));
47     return;
48  }
```

将文件保存为 exam611printbuf.c，在终端中使用 gcc 对其进行编译链接，生成可执行文件 exam611printbuf。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam611printbuf.c -o exam611printbuf
```

在当前工作目录下执行该可执行文件，可以看到对应的输出。

```
alloy@ubuntu:~/linuxc/chapter6$ sudo ./exam611printbuf
当前流是 stdin, 全缓冲, 缓冲区大小 = 0
当前流是 stdout, 行缓冲, 缓冲区大小 = 1024
当前流是 stderr, 无缓冲, 缓冲区大小 = 0
```

2. 设置流缓冲区的大小

如果需要对 Linux 内核提供的流缓冲状态进行修改，可以使用 setbuf 系列函数，对其标准调用

格式说明如下：

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
void setbuffer(FILE *stream, char *buf, size_t size);
void setlinebuf(FILE *stream);
```

这 4 个函数都必须在流成功打开之后再调用，setbuf 函数没有返回值，如果 setvbuf 函数调用成功，则返回“0”，否则返回一个非“0”，对其参数说明如下。

- stream 参数：流指针，指向一个打开的流。
- buf 参数：在 setbuf 函数中指向一个长度为 BUFSIZ 的缓冲区，BUFSIZ 是在 stdio.h 中定义的一个宏，其长度为 8192 字节；在 setvbuf 函数中，缓冲区的大小由 size 确定。
- mode 参数：缓冲类型，包括_IOFBF（全缓冲）、_IOLBF（行缓冲）和_IONBF（不带缓冲）。
- size 参数：缓冲区的大小。

对于 setbuf 函数来说，其不存在 mode 和 size 参数，所以其设定的流通常是全缓冲的，若这个流和终端设备相关，则有可能设置为行缓冲，同时可以通过将 buf 设置为 NULL 来关闭缓冲。

对于 setvbuf 函数来说，通过设置 mode 和 size 可以选择相应的缓冲方式和缓冲区大小，如果设置一个流带缓冲，但是 buf 被设置为 NULL，则 Linux 内核会自动将其缓冲区大小设置为 BUFSIZ。表 6.5 是以上两个函数的总结。

表 6.5 setbuf 和 setvbuf 总结

<table>
<tr><th>函数</th><th>mode</th><th>Buf</th><th>缓冲区及其长度</th><th>缓冲类型</th></tr>
<tr><td rowspan="2">setbuf</td><td rowspan="2"></td><td>非空</td><td>用户缓冲区，长度 BUFSIZ</td><td>全缓冲或者行缓冲</td></tr>
<tr><td>NULL</td><td>无缓冲区</td><td>不带缓冲</td></tr>
<tr><td rowspan="5">setvbuf</td><td rowspan="2">_IOFBF</td><td>非空</td><td>用户缓冲区，长度为 size</td><td rowspan="2">全缓冲</td></tr>
<tr><td>NULL</td><td>系统缓冲区，长度通常为 BUFSIZ</td></tr>
<tr><td>_IOLBF</td><td>非空</td><td>用户缓冲区，长度为 size</td><td rowspan="2">行缓冲</td></tr>
<tr><td rowspan="2">_IONBF</td><td>NULL</td><td>系统缓冲区，长度通常为 BUFSIZ</td></tr>
<tr><td>-</td><td>无缓冲区</td><td>不带缓冲</td></tr>
</table>

setbuffer 函数将流设置为全缓冲方式，其可以指定缓冲区的大小。setlinebuf 函数是将流设置为行缓冲方式。



注 意

最好在将流打开但还未对流执行其他操作时设定流的属性，因为对流的各种操作都是和缓冲区的属性紧密相关的，改变缓冲区的属性会对所执行的操作产生意想不到的影响。

例 6.12 是使用 setvbuf 函数对缓冲区进行设置的实例。

【例 6.12】使用 setvbuf 函数设置缓冲区的大小

应用代码首先使用setvbuf函数将标准输入stdin设置为无缓冲，然后在屏幕上打印其缓冲类型，然后将其设置为全缓冲，缓存大小为 512 字符，再次打印其缓冲类型，流程如图 6.10 所示。

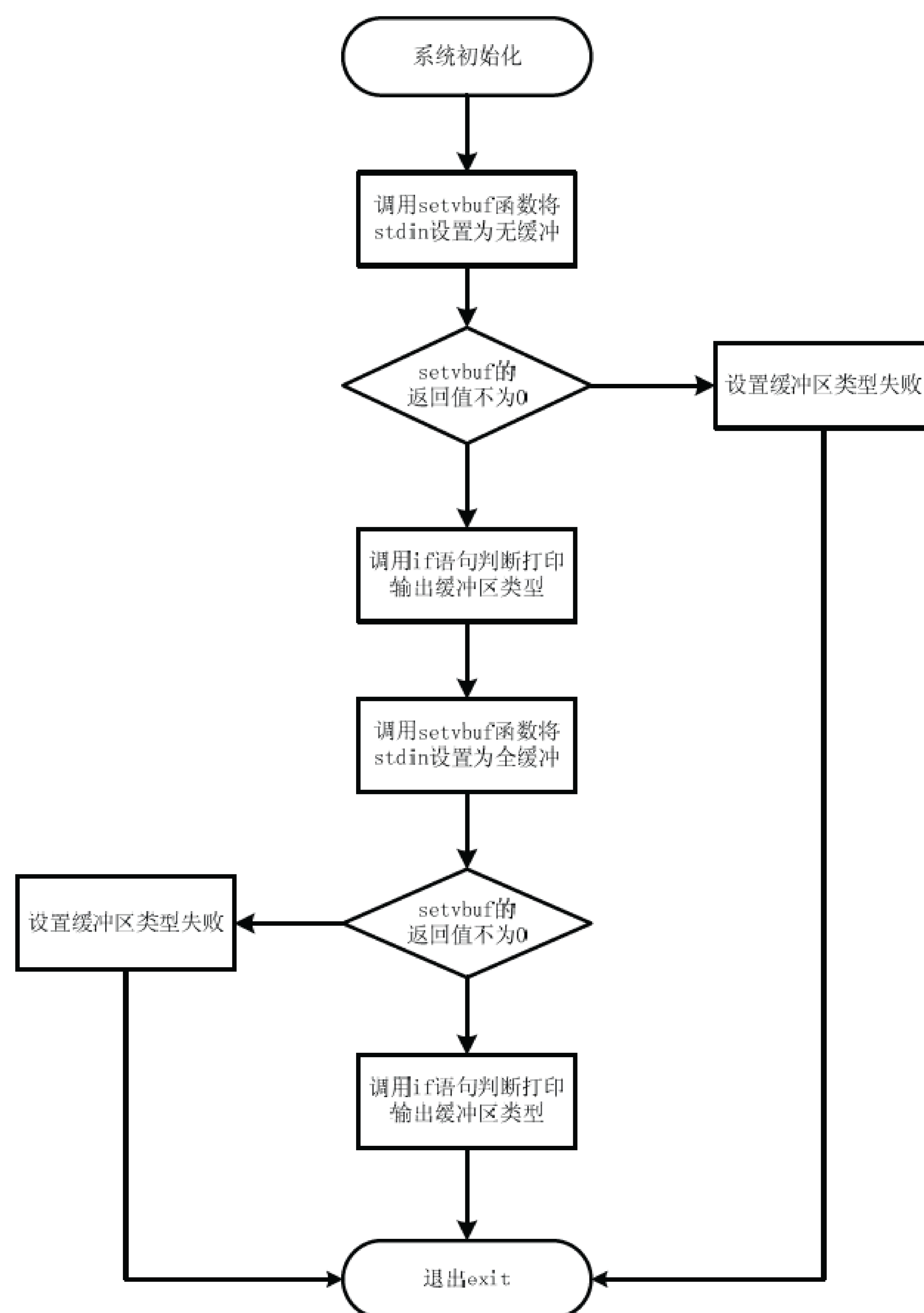


图 6.10　使用 setvbuf 设置缓冲区大小

实例的应用代码如下：

```
1   //流操作的缓冲区设置应用实例
2   //调用 setbuf 函数来修改标准输入 stdin 的缓冲方式
3   #include <stdio.h>
4   #include <stdlib.h>
5   #define SIZE 512                                   //定义缓冲区大小
6   int main(int argc,char *argv[])
7   {
8       char buf[SIZE];                                //缓冲区
9       if(setvbuf(stdin, buf, _IONBF, SIZE) != 0)     //将标准输入的缓冲类型设为无缓冲
10      {
11          perror("将标准输入 stdin 的输入设置为无缓冲失败!\n");          //如果设置失败
```

```
12              return 1;
13          }
14          printf("将标准输入 stdin 的输入设置为无缓冲成功!\n");
15          printf("stdin 类型为");                          //打印缓冲区信息
16          if(stdin->_flags & _IO_UNBUFFERED)               //判断标准输入流对象的缓冲区类型
17          {
18              printf("无缓冲\n");
19          }
20          else if(stdin->_flags & _IO_LINE_BUF)
21          {
22              printf("行缓冲\n");
23          }
24          else
25          {
26              printf("全缓冲\n");
27          }
28          printf("缓冲区大小为 %ld\n", stdin->_IO_buf_end - stdin->_IO_buf_base);
29          //打印缓冲区的大小
30          printf("文件描述符为 %d\n", fileno(stdin));          //输出文件描述符
31          if(setvbuf(stdin,buf,_IOFBF,SIZE)!=0)
32          {
33              //将标准输入的缓冲类型设为全缓冲，缓存大小为 512MB
34              printf("将标准输入 stdin 设置为全缓冲失败!\n");
35              return 2;                                    //出错退出
36          }
37          printf("修改标准输入 stdin 的类型成功!\n");
38          printf("stdin 类型为");                          //打印缓冲区信息
39          if(stdin->_flags & _IO_UNBUFFERED)               //判断标准输入流对象的缓冲区类型
40          {
41              printf("无缓冲\n");
42          }
43          else if(stdin->_flags & _IO_LINE_BUF)
44          {
45              printf("行缓冲\n");
46          }
47          else
48          {
49              printf("全缓冲\n");
50          }
51          printf("缓冲区大小为%ld\n", stdin->_IO_buf_end - stdin->_IO_buf_base);
52          //打印缓冲区的大小
53          printf("文件描述符为%d\n", fileno(stdin));           //输出文件描述符
54          return 0;
55      }
```

将文件保存为 exam612setbuf.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam612setbuf。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam612setbuf.c -o exam612setbuf
```

运行该可执行文件，可以看到对应的设置结果，标准输入首先被设置为无缓冲类型，此时缓冲区大小为 1，然后被设置为全缓冲类型，缓冲区大小为 512；而其对应的文件描述符则是固定的、不会发生变化的 0。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam612setbuf
```

```
将标准输入 stdin 的输入设置为无缓冲成功!
stdin 类型为无缓冲
缓冲区大小为  1
文件描述符为  0
修改标准输入 stdin 的类型成功!
stdin 类型为全缓冲
缓冲区大小为 512
文件描述符为 0
```

6.2.6　流的冲洗

在使用流的时候，在系统内核中开辟了一块缓冲区用于相应的操作，在关闭流或者操作完成之后，应该将缓冲区的数据清空，这种清空可以是将流的内容完全丢掉，也可以是将其保存到流对应的文件中，这个过程叫做流的冲洗。Linux 内核同样提供了相应的函数 fflush 和_fpurge 来完成流的冲洗，对其标准调用格式如下：

```
#include <stdio.h>
int fflush(FILE *stream);
#include <stdio.h>
#include <stdio_ext.h>
void    __fpurge(FILE *stream);
```

fflush 将参数 stream 指定流的缓冲区中尚未写入文件的数据强制性地保存到文件中，如果调用成功，则返回值为 0；若调用失败则返回 EOF。

_fpurge 函数用于将缓冲区中的数据完全清除，由于使用较少，因此这个函数定义在<stdio_ext.h>中。


注　意

流的冲洗在调用 fclose 函数时关闭这个流，或者一个进程使用 exit、return 函数来正常终止的时候是会自动进行的，并不需要用户特意进行，但是如果有其他特定的需求也可以由用户调用以上的函数来手动完成。

6.3　流的格式化输入和输出

除了按字符、行和二进制的流输入输出方式之外，还可以使用流的格式化输入输出方式，其特点是可以将输入输出规格化为一定的组成结构，然后输出。

putc、gets 等 I/O 函数除了将数据分解成字符或者行之外，并不对所操作的数据进行解释，但有时对数据进行解释却是必要的，因为在 Linux 内核中数据的表示与用户习惯的阅读形式不同，例如，十进制数 10 在计算机内部的 32 位表示是：

```
00000000000000000000000000001010
```

但是，当这个数在打印机上输出或者在终端屏幕上显示时，它必须转换为 ASCII 字符“1”和“0”，这 2 个字符在计算机内部有完全不同的表示：

```
1：00110001
0：00110000
```

类似的，为了从键盘读入十进制整数 10，将十进制数转换为计算机可处理的内部表示。

格式化 I/O 函数能够自动完成这种数据外部格式和内部格式的转换工作，并且能够对输入输出数据进行诸如数据类型、精度、位置等格式控制，它们是标准 I/O 库中使用最频繁的函数。

所有格式化 I/O 函数的调用界面都是简单的，它们都通过一个格式字符串来对其余参数进行格式描述。但是，格式字符中的转换区分符由子格式本身的复杂性而五花八门，因为它们要描述每一种数据类型（整数、浮点数、十进制数、八进制数、十六进制数、字符、字符串……），数据的精度（单精度、双精度、短整数、长整数……），数据的外部形式（指数形式、定点形式、左对齐、右对齐、是否有前缀 0、是否有正号或负号……）以及字节宽度等等。

6.3.1 流的格式化输出

Linux 内核提供了 4 个 printf 函数用于流的格式化输出，对其标准调用格式说明如下：

```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

printf 函数用于将格式化的数据写入到标准输出（在第 3 章中已经有过介绍），fprintf 函数则用于将格式化的数据写入到一个由 stream 指针指向的流，如果这两个函数操作成功则返回输出的字符数，如果操作失败则返回一个负值。

sprintf 函数用于将格式化的数据写入到 str 指向的缓冲区数组并且在末尾加上一个 NULL，而 snprintf 则可以使用 size 参数来指定这个缓冲区数组的大小，超过该大小的数据将被丢弃，如果这两个函数操作成功则返回存入数组的字符数，如果编码出错则返回一个负值。

这 4 个函数的 format 参数用于说明格式化数据的格式化方法，将在第 5.6.3 小节中进行介绍。


注 意

Linux 内核还提供了 vprintf 等 4 个对应的变体函数，读者可以自行查阅相应的帮助手册。

6.3.2 流的格式化输入

和 printf 系列函数对应，Linux 内核同样提供了相应的 scanf 系列格式化输入函数对输入的字符串进行分析并且将转化为指定类型的变量，格式化之后的参数包括了对应的变量地址，可利用转换结果来初始化这些变量，对其标准调用格式说明如下：

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

scanf 函数用于从标准输入设备中按照 format 提供的格式读取数据，fscanf 用于从一个流按照

format 提供的格式读取数据，而 sscanf 则从 str 指定的字符缓冲区中读取数据，如果调用成功则返回输入的项数，如果输入出错或者到达文件结尾则返回 EOF。


注　意

Linux 内核同样提供了 vscanf 系列变体函数以供调用。

6.3.3　流的格式化参数

对 printf 系列函数的格式化参数 format 的标准组成部分说明如下：

```
%[flags][fldwidth][precision][lenmodifier]convtype
```

表 6.6 给出了 format 参数的详细说明。

表 6.6　format 参数详细说明

选项	说明
%	“%”为一个范围为[1~NL_ARGMAX]的十进制正整数，它指出参数表中下一个要输出的参数位置（紧接在 format 参数之后那个参数的位置为 1）。这个修饰符使得格式字符串能够按任意顺序选择要输出的参数和多次输出同一参数
flags	称为标志，由 0 至多个标志字符按任意顺序组成，所允许的标志字符及其含义如下： ● ': 十进制转换（i，d，u，f，g 或 G）结果的整数部分按千位分组格式打印，转换结果在域中左对齐。当无此标志字符时，为右对齐 ● +: 转换结果总是带有符号(+或-)。当无此标志字符时，只有负数才带有负号(-) ● 空格：对有符号类型的数据，若结果不是以“+”或“-”开头，则用空格字符作为其前缀。因为“+”标志保证结果含有一个符号，因此如果同时指定这两个标志，则本标志被忽略 ● #: 指定参数按选择格式转换，具体视转换字符而定 ● 0: 用打头 0 替代空格填充域宽，如果同时指定“0”和“-”标志，则忽略 0 标志。对于整型转换 d，i，o，u 和 X，如果指定了精度，则 0 标志将被忽略。如果同时指定 0 和“'”标志，则在填充的打头 0 之间插入千位分组符
fldwidth	一个十进制正整数或者“*”，称为宽度指示符，用于指明最小域宽，域宽，即结果所占的字符位置数
precision	一个十进制正整数或者“*”，称为精度指示符，用于指明最小域宽，域宽，即结果所占的字符位置数
lenmodifier	称为长度修饰符，当没有长度修饰符时，对应的参数处理成 int 类型（对于转换 i 和 d）、unsigned int 类型（对于转换 o，u，x 和 X）或者 double 类型（对于 e，E，f，g 和 G 转换）。任何类型为 char 或 short 的参数将自动转换为 int 类型。当要输出的参数类型长度与默认类型长度不同时需要使用长度修饰符

和 printf 系列函数类似，scanf 函数除了是从输入流读取数据并存储值至对应参数所指的地址中之外，其也用类似的方式使用格式字符串 format 来控制格式转换，并且许多转换区分行也是相

同的，但其与 printf 函数有两点重要的不同：

- 虽然输入转换区分符的语法与 printf 函数中的语法类似，但它们的解释主要是面向自由格式的输入和简单的模式匹配，而不是针对格式化的固定域。例如，大部分 scanf 转换都跳过输入文件中的空白字符（包括空格符、制表符、换行符等），并且对于数值转换没有对应输出转换那样的精度概念。
- scanf 系列函数中位于 format 参数之后的所有其他参数都应当是指针，所读入的值将存储在指针所指对象之中。

scanf 系列函数使用的格式字符串 format 由 3 类成分组成：

- 一至多个连续的空白符：包括空格符“ ”、制表符“\t”、水平制表符“\v”、换行符“\r”和走纸符“\f”，对于这种类型的字符 scanf 系列函数将直接跳过一直到遇到一个未读过的非空白字符或者遇到文件尾，需要注意的是输入流中的空白字符不必完全与格式字符串中的空白符相同。
- 普通字符（不包括“%”和空白符）：用于指明必须出现在输入流中的字符，它必须完全与输入流中的下一字符相匹配，如果不匹配将导致匹配失败。
- 转换区分符：格式字符串中的转换区分符指导下一个输入域的转换。输入域定义为输入流中非空白字符组成的字符序列，其长度直至遇到一个不合适的字符或者到达指定的域宽为止。转换后的结果存储在对应的参数中，除非转换区分符指明了禁止赋值标志“*”。大部分转换区分符通常都忽略输入中的空白字符，这意味着%d 将一直读输入直至发现一个数字序列。如果所期望的字符没有出现，该转换将失败且 scanf 将立即返回。


注 意

一定要区分术语“空白符”、“空格符”和“空字符”。空白符包括空格符“ ”、制表符“\t”、水平制表符“\v”、换行符“\r”和走纸符“\f”，即 isspace 函数返回值为真的字符；空格符是指“ ”；空字符是指 null（即“\0”）字符。

对 scanf 系列函数的格式化参数 format 的标准组成部分说明如下：

```
%[*][fldwidth] [lenmodifier]convtype
```

表 6.7 是 scanf 函数的 format 格式说明。

表 6.7 format 格式说明

修饰符	说明
%	“%”为一个范围为[1~NL_ARGMAX]的十进制正整数，它指出参数表中下一个要输出的参数位置（紧接在 format 参数之后的那个参数位置为 1）。这个修饰符使得格式字符串能够按任意顺序选择要赋值的参数和多次对同一参数赋值
*	禁止赋值标志，使得 scanf 忽略所读出的输入值，但不使用指针参数并且也不增加成功赋值计数

（续表）

修饰符	说明
fldwidth	一个十进制正整数，称为宽度指示符，用于指明最大域宽。当到达此最大域宽或发现一个非匹配字符时，不论谁先发生，均停止读输入流。大多数转换字符均忽略打头的空白字符，这些被忽略的空白符在此最大域宽内不计数。字符串转换存储一个空字符作为该输入的结束，最大域宽也不包括这个终止符
lenmodifier	称为长度修饰符，指明接收对象的类型长度。当没有长度修饰符时，对应的参数处理成 int 类型（对于转换 i 和 d）、unsigned int 类型（对于转换 o，u，x 和 X）或者 float 类型（对于 e，E，f，g 和 G 转换）。当接收对象的类型长度与默认类型长度不同时需要使用如下长度修饰符。 ● H：对于 i，d，和 n 转换，指明参数是一个 short int *；对于 o，u 和 x 转换，指明参数是一个 unsigned short int * ● l：对于 i，d 和 n 转换，指明参数是一个 long int *；对于 o，u 和 x 转换，指明参数是一个 unsigned short int * ● L：对于浮点转换，指明参数是一个 long double *

表 6.8 是各种输入转换字符类型的说明。

表 6.8　输入转换字符说明

选项	说明
d	匹配一个十进制形式的有符号整数
i	匹配一个 C 语言为任何形式的有符号整数
o	匹配一个八进制形式的无符号整数
u	匹配一个十进制形式的有符号整数
x，X	匹配一个十六进制形式的无符号整数，x 用小写字母，X 用大写字母
f，e，E，g，G	匹配任意一个有符号浮点数，它们之间可以互相替换
c	匹配单个字符或由多个字符组成的字符串，读入的字符个数由最大域宽指示符指定或默认为 1。它不在读入的正文之后附加一个空字符，也不跳过打头的空白符。它严格地读域宽给定的 n 个字符，并且当不能达到这么多个字符对时将导致失败
s	匹配一个由空白字符组成的字符串。它跳过并丢弃打头的空白字符，仅在读入非空白字符之后停止于遇到的第一个空白字符。它在读入的正文尾部附加一个空字符
[	这种转换包括所有后续字符直至相匹配的右方括弧“]”。它匹配一个由特定字符集中的字符组成的字符串。这个特定字符集使用与正则表达式相同的语法，定义于“[”和“]”之间，但有如下特殊情形： ● 如果这个特定集合中也包括字符“]”，则它必须是紧跟在初始“[”之后的第一个“]”字符，与初始“[”相匹配的右方括弧将是下一个“]” ● 嵌入在正则表达式内的字符“-”（既不是第一个字符，也不是最后一个字符）用于指明字符的范围 ● 紧跟在初始“[”之后的脱字符“^”指明所允许的字符集合是除列出的字符之外的任何字符
p	用于读一指针值。它所识别的语法同 printf 的%p 输出转换相同，即一个恰如%x 转换接收的十六进制数。对应的参数必须是 void **类型，即一个存放指针的地址
n	这一转换不读任何字符，它记录此次调用中迄今为止已读入的字符数
%	匹配输入流中的字符“%”，这个转换不使用参数

6.3.4 流的格式化输入输出操作实例

例 6.13 和例 6.14 是两个流的格式化输入输出操作实例。

【例 6.13】使用 fprintf 函数将字符串写入文件

应用代码定义了三个 int 类型的变量 h、m、s，分别对应当前时间的时、分、秒信息，使用 fprintf 函数将其格式化，并输出到 argv 指定的文件中，其流程如图 6.11 所示。

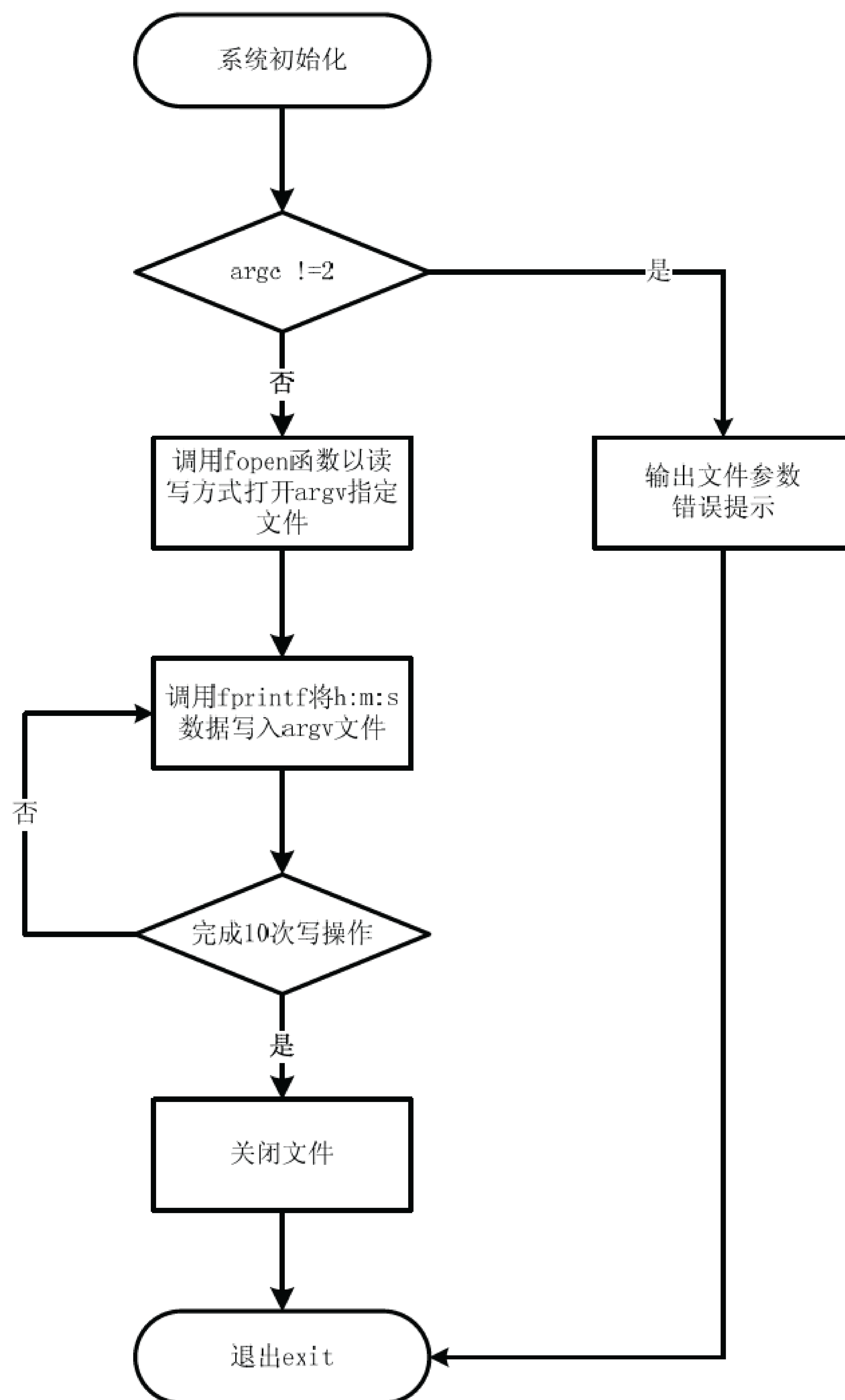


图 6.11 使用 fprintf 函数将字符串写入文件

实例的应用代码如下：

```
1  //将一个格式化的字符串写入文件
2  #include <stdio.h>
3  int main(int argc,char *argv[])
```

```
 4  {
 5     FILE *fp;                           //流文件结构指针
 6     int h,m,s;                          //时分秒信息
 7     int temp;                           //存放 fprintf 的返回值
 8     int i;
 9     if(argc != 2)                       //文件参数错误
10     {
11         printf("文件参数错误\n");
12         return 1;
13     }
14     h = 9;
15     m = 51;
16     s = 19;
17     fp = fopen(*(argv+1),"a+b");        //读写方式打开文件
18     for(i = 0;i<10;i++)
19     {
20         temp = fprintf(fp,"%02d%02d%02d\n",h,m,s);          //打印字符串到 fp 中
21         if(temp < 0)                                        //打印出错
22         {
23             printf("第%d 次将字符串打印到%s 文件中失败\n",i,*(argv+1));
24             return 2;
25         }
26         else
27         {
28             printf("第%d 次将%d 个字符打印到%s 文件成功\n",i,temp,*(argv+1));
29         }
30     }
31     fclose(fp);                                             //关闭流
32     return 0;
33  }
```

将文件保存为 exam613fprintf.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam613fprintf。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam613fprintf.c -o exam613fprintf
```

在当前工作路径下执行可执行文件 exam613fprintf，将产生的字符写入一个新的文本文件 fprintftest.txt 中，然后使用“cat –n”命令查看该文件内容，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam613fprintf fprintftest.txt
第 0 次将 7 个字符打印到 fprintftest.txt 文件成功
第 1 次将 7 个字符打印到 fprintftest.txt 文件成功
……    //其间省略部分内容
第 9 次将 7 个字符打印到 fprintftest.txt 文件成功
alloy@ubuntu:~/linuxc/chapter6$ cat fprintftest.txt -n
 1   095119
 2   095119
……          //器件省略了部分内容
 9   095119
```

```
10   095119
```

使用“cat –n”查看 fprintftest.txt 可以知道在例 6.13 中写入文件的字符串都是固定的，如果想产生不同的字符串，可以使用 rand 和 srand 函数来产生相应的随机数。

rand 函数的标准调用格式如下：

```
#include<stdlib.h>
int    rand (void);
```

调用成功之后返回一个 0~RAND_MAX 的整型数据，其中 RAND_MAX 在 stdlib.h 头文件中有定义，默认是 2147483647。

需要注意的是，rand 函数的内部实现是用线性同余进行处理的，其并不是真的随机数，只不过是因为其周期特别长，所以在一定的范围里可看成是随机的。

在调用此函数产生随机数前，必须先调用 srand 函数来初始化随机数种子，如果未设随机数种子，则 rand 函数在调用时会自动设随机数种子为 1。另外一个需要注意的是 rand 函数产生的是假随机数字，每次执行时是相同的；若要不同，则需要调用 srand 函数利用不同的随机数种子来初始化该函数。

srand 函数的标准调用格式如下，其中 seed 为初始化参数，srand 函数没有返回值。

```
#include <stdlib.h>
void srand(unsigned int seed);
```


注 意

在 Linux 系统中，通常使用 getpid()或者 time(0)的返回值作为 srand 函数的参数。

此时可以把例 6.13 的应用代码修改为如下形式：

```
1   //将一个格式化的字符串写入文件
2   #include <stdio.h>
3   #include <stdlib.h>
4   int main(int argc,char *argv[])
5   {
6       FILE *fp;                              //流文件结构指针
7       int h,m,s;                             //时分秒信息
8       int temp;                              //存放 fprintf 的返回值
9       int i;
10      if(argc != 2)                          //文件参数错误
11      {
12          printf("文件参数错误\n");
13          return 1;
14      }
15      srand((int)time(0));                   //调用 srand 函数对随机数函数 rand 的种子进行初始化
16      fp = fopen(*(argv+1),"a+b");           //读写方式打开文件
17      for(i = 0;i<10;i++)
18      {
19          h = 1 + (int)(10.0 * rand()/RAND_MAX + 1.0);
```

```
20          m = 1 + (int)(10.0 * rand()/RAND_MAX + 1.0);
21          s = 1 + (int)(10.0 * rand()/RAND_MAX + 1.0);
22          //分别产生 3 个位于 1~10 的随机数
23          temp = fprintf(fp,"%02d%02d%02d\n",h,m,s);          //打印字符串到 fp 中
24          if(temp < 0)                                        //打印出错
25          {
26              printf("第%d 次将字符串打印到%s 文件中失败\n",i,*(argv+1));
27              return 2;
28          }
29          else
30          {
31              printf("第%d 次将%d 个字符打印到%s 文件成功\n",i,temp,*(argv+1));
32          }
33      }
34      fclose(fp);                                             //关闭流
35      return 0;
36  }
```

将文件保存为 exam613randfprintf.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam613randfprintf。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam613randfprintf.c -o exam613randfprintf
```

执行该可执行文件生成一个新的文件 randfprintf.txt，然后使用“cat –n”命令查看该文件内容，可以看到每次写入的字符串都不同。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam613randfprintf randfprintf.txt
第 0 次将 7 个字符打印到 randfprintf.txt 文件成功
第 1 次将 7 个字符打印到 randfprintf.txt 文件成功
第 2 次将 7 个字符打印到 randfprintf.txt 文件成功
第 3 次将 7 个字符打印到 randfprintf.txt 文件成功
第 4 次将 7 个字符打印到 randfprintf.txt 文件成功
第 5 次将 7 个字符打印到 randfprintf.txt 文件成功
第 6 次将 7 个字符打印到 randfprintf.txt 文件成功
第 7 次将 7 个字符打印到 randfprintf.txt 文件成功
第 8 次将 7 个字符打印到 randfprintf.txt 文件成功
第 9 次将 7 个字符打印到 randfprintf.txt 文件成功
alloy@ubuntu:~/linuxc/chapter6$ cat randfprintf.txt -n
 1   020908
 2   070410
……                                                  //其间省略部分内容
 9   110405
10   071006
```

【例 6.14】使用 scanf 函数读取数据

应用代码先创建一个 argv 参数指定的新文件，如果创建成功则使用 fprintf 函数输入 4 个格式数据，然后定位到文件的开头，再使用 fscanf 函数逐个读取出来发送至屏幕显示，其流程如图 6.12 所示。

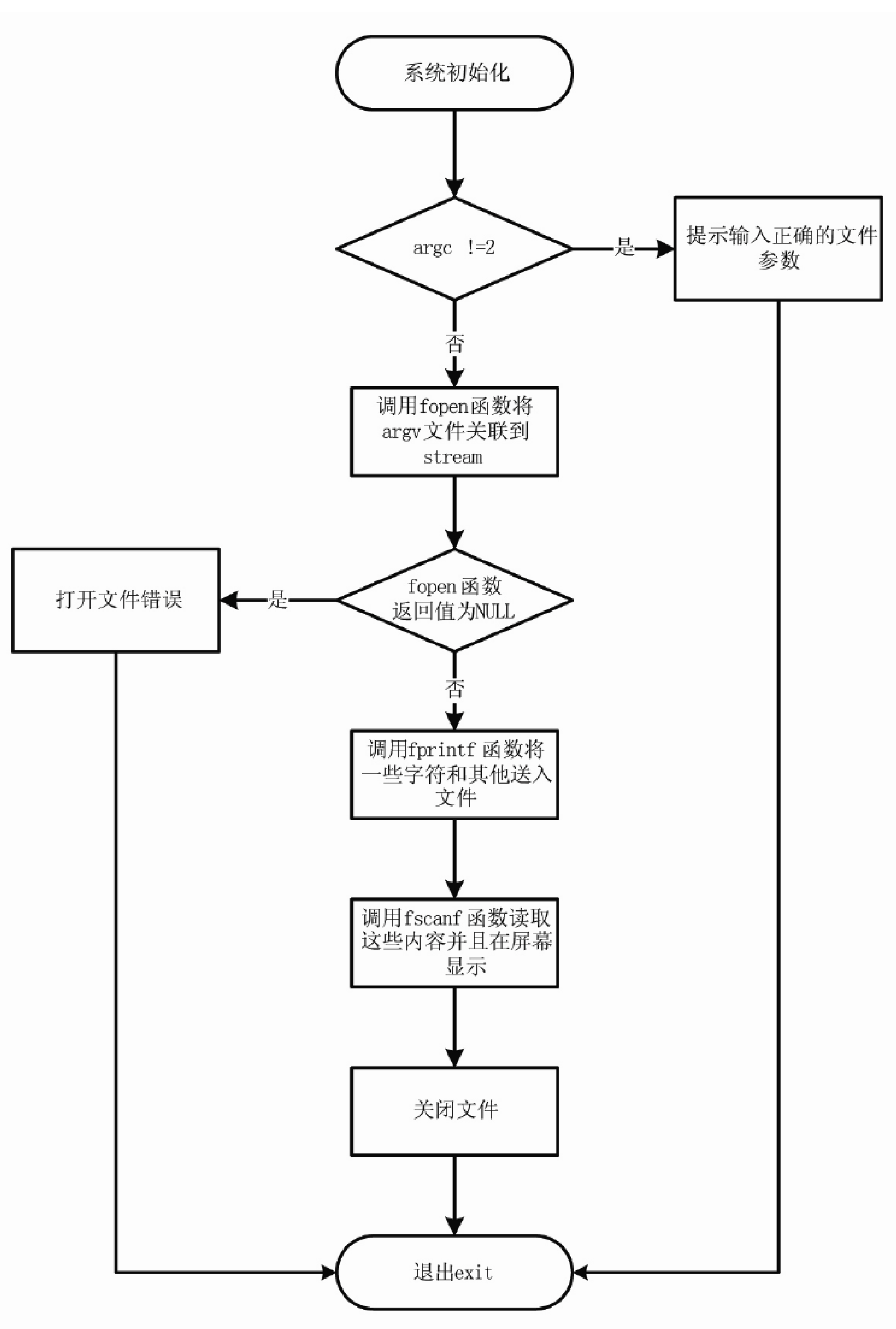


图 6.12　使用 fscanf 函数读取数据

实例的应用代码如下：

```
1   /*函数 fscanf()示例*/
2   #include<stdio.h>
3   FILE* stream;
4   int main(int argc,char *argv[])
5   {
6           long l;
7           float fp;
8           char s[81];
9           char c;
10          if(argc != 2)
```

```
11          {
12             printf("文件参数错误！ \n");
13             return 0;
14          }
15          stream = fopen(*(argv+1),"w+");
16          if(stream == NULL)
17          {
18                printf("打开文件失败!\n");
19          }
20          else
21          {
22                fprintf(stream,"%s %d %f %c","a_string",6500,3.1415,   'x');
23                fseek(stream,0L,SEEK_SET);                    /*定位文件*/
24                fscanf(stream,"%s",s);                        /*格式化*/
25                fscanf(stream,"%ld",&l);
26                fscanf(stream,"%f",&fp);
27                fscanf(stream," %c",&c);
28                printf("%s\n",s);
29                printf("%ld\n",l);
30                printf("%f\n",fp);
31                printf("%c\n",c);
32                fclose(stream);/*关闭*/
33          }
34          return 0;
35    }
```

将文件保存为 exam614scanf.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam614scanf。

```
alloy@ubuntu:~/linuxc/chapter6$ gcc exam614scanf.c -o exam614scanf
```

调用 exam614scanf 文件对一个 scanftest.txt 文件进行操作，可以看到如下的输出，这些字符都是先被写入 scanftest.txt 文件中，然后被 fscanf 函数读出的。

```
alloy@ubuntu:~/linuxc/chapter6$ ./exam614scanf scanftest.txt
a_string
6500
3.141500
x
```

6.4　本章习题

1. 编写一个程序，从键盘输入一个字符，并将其显示出来，当输入 q 时，程序退出。
2. 编写一个程序，查看本机中标准输入流的缓冲区类型
3. 编写一个程序，从键盘中输入字符，并将它写入一个文件，当输入 q 时，程序退出。
4. 编写一个程序，利用 sprintf 函数把二进制数据转换为十进制字符串的形式。
5. Linux 中的 wc 命令的功能为统计指定文件中的字节数、字数、行数，并将统计结果显示输

出，编写一个程序，对标准输入设备（键盘）实现 wc 命令的功能。

6. 在某文件中将学生的各种信息都存放在一个如下的结构体中：

```
struct {
        char name[NAMESIZE];
        long number;
        short department;
        short scores[10];
} student;
```

编写一个程序，将存放学生各种信息的文件中的学生信息读出，重新组成一个存放所有学生的前 3 门成绩的文件。

第 7 章　Linux 的进程

进程（Process）是多任务操作系统的基础概念，其通常被定义为程序执行时的一个实例，是可以分配给处理器并由处理器执行的一个实体，是由单一顺序的执行显示，是一个当前状态和一组相关的系统资源所描述的活动单元。本章将详细介绍 Linux 进程的基础知识以及对其的操作方法。

7.1　操作系统和进程

支持多任务的操作系统在执行多个任务的时候需要共享系统资源，从而导致各程序在执行过程中出现相互制约的关系，程序的执行表现出间断性的特征。这些特征都是在程序的执行过程中发生的，是动态的过程，而 C 语言程序编译链接后生成的可执行文件本身是一组指令的集合，是一个静态的概念，无法描述程序在内存中的执行情况，即无法从程序的字面上看出它何时执行，何时停顿，也无法看出其与其他执行程序的关系，因此，程序这个静态概念已不能如实反映程序并发执行过程的特征，为了深刻描述程序动态执行过程的性质，引入了进程（Process）这个概念。

7.1.1　进程的特点

进程是上个世纪 60 年代初首先由麻省理工学院的 MULTICS 系统和 IBM 公司的 CTSS/360 系统引入的。其是一个具有独立功能的程序关于某个数据集合的一次运行活动，其可以申请和拥有系统资源，是一个动态的概念，是一个活动的实体，不只是程序的代码，还包括当前的活动，通过程序计数器的值和处理寄存器的内容来表示。

进程由程序、数据和进程控制块三部分组成，多个不同的进程可以包含相同的可执行文件，一个可执行文件在不同的数据集里构成不同的进程，得到不同的结果，但是在执行过程中，其不能发生改变。例如在第 3 章的 open 函数应用实例 3.4 中生成的可执行文件 exam302openFun 可以用于打开/创建一个文件，当其运行的时候即形成了一个进程，而在 Linux 中可以“同时”调用两次该执行文件，即形成了两个进程，可以“同时”创建两个文件。

进程具有如下几个特点。

- 动态性：进程的实质是一个可执行文件（程序）在多任务操作系统中的一次执行过程，所以其是动态产生和消亡的，也就是说 exam302openFun 在运行时成为一个进程，当创建文件完成之后会消亡，该进程消失。
- 并发性：在多任务操作系统中任何进程都可以同其他进程一起并发执行，也就是说 exam302openFun 可以和其他的执行文件包括自身一起并发执行。
- 独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位，当 exam302openFun 运行成为进程之后 Linux 系统会给其分配对应的内存等资源，这些

资源是该进程独立拥有的，不会被其他进程所使用。

- 异步性：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进。也就是说当一个进程需要进行多步操作的时候，首先其并不一定是连续执行的，由于计算机的处理器等资源是有限的，其可能需要等待其他进程使用这些资源；其次当多个进程同时运行的时候，这些进程的第 1 步、第 2 步、第 3 步……第 n 步操作并不是一一对应执行的。

7.1.2 进程和可执行文件（程序）的区别

进程和可执行文件（程序）是两个完全不同的概念，其主要区别如下。

- 当同一个可执行文件（程序）同时运行于若干个数据集合上，其将属于若干个不同的进程，也就是说同一可执行文件（程序）可以对应多个进程，例如用户可以同时执行多个 exam302openFun，此时每个运行中的 exam302openFun 对应一个进程。
- 可执行文件（程序）是指令和数据的有序集合，其本身没有任何运行的含义，是一个静态的概念；而进程是程序在计算机上的一次执行过程，它是一个动态的概念。当 exam302openFun 不被用户运行的时候，其不会有任何动作，而当其开始运行，启动一个进程的之后该进程将自动运行，创建/打开一个指定文件，并且退出。
- 可执行文件（程序）可以作为一种数据资料长期存在，是永久的；而进程是有一定生命期的，是暂时的。对于 exam302openFun 来说只要用户不将其从存储空间中删除则会一直存在，而其对应的进程在创建/打开一个指定文件之后会自动退出，此时进程也不存在了。
- 可执行文件（程序）不存在并发概念；而进程能真实地描述并发。
- 可执行文件（程序）是由存储器空间中的若干可执行代码组成；而进程是由进程控制块、程序段、数据段三部分组成，这部分内容将在下一小节中介绍。
- 可执行文件（程序）由于是静止的，其并不能创建其他可执行文件（程序）或者进程；而进程具有创建其他可执行文件（程序）和进程的能力。

7.2 Linux 的进程基础

Linux 是一个多用户多任务的操作系统，多用户是指多个用户可以在同一时间使用同一台计算机系统；多任务是指 Linux 可以同时执行几个任务，它可以在还未执行完一个任务时又执行另一项任务，Linux 内核管理着多个用户的请求和多个任务，每个任务或者请求都对应着一个进程。

7.2.1 Linux 进程的基础属性

Linux 系统上所有运行的任务都可以是一个进程，每个用户任务、每个系统管理，都可以称之为进程，Linux 用分时管理的方法使所有的任务共同分享系统资源。当讨论进程的时候，不会去关心这些进程究竟是如何分配的，或者是内核如何管理分配时间片的，用户所关心的是如何去控制这些进程，让它们能够很好地为自己服务。

对于 Linux 中的进程，一个比较正式的定义是：在自身的虚拟地址空间运行的一个单独程序。进程与程序是有区别的，进程是动态的，程序是静态的，进程不是程序，虽然它由程序产生。程序只是一个静态的命令集合，不占系统的运行资源；而进程是一个随时都可能发生变化的、动态的、使用系统运行资源的程序，而且一个程序可以启动多个进程。

1. 进程的 4 个要素

在 Linux 中，一个进程必须具有以下 4 个要素：

- 要有一段程序代码以供该进程运行。
- 拥有专用的系统堆栈空间。
- 拥有一个由 task_struck 结构来实现进程控制块。
- 拥有独立的存储空间。

2. 进程的关系和分类

Linux 系统中的所有进程都是相互联系的，程序创建的进程之间具有父/子关系，而自进程之间具有兄弟关系。

Linux 内核创建了进程标号为 0 以及进程标号为 1（关于进程标号将在下一小节进行介绍）的进程，其中进程标号为 1 的进程是一个初始化进程 init，Linux 中的所有进程都是由其衍生而来的，在 Shell 下执行程序启动的进程则是 Shell 进程的子进程，在用户的启动进程中可以再启动自己的子进程，这样就形成了一棵进程树，每个进程都是树中的一个节点，其中树的根是初始化进程 init。

通常来说进程之间的关系可以用如图 7.1 所示的亲属关系来描述，通常包括以下几个部分。

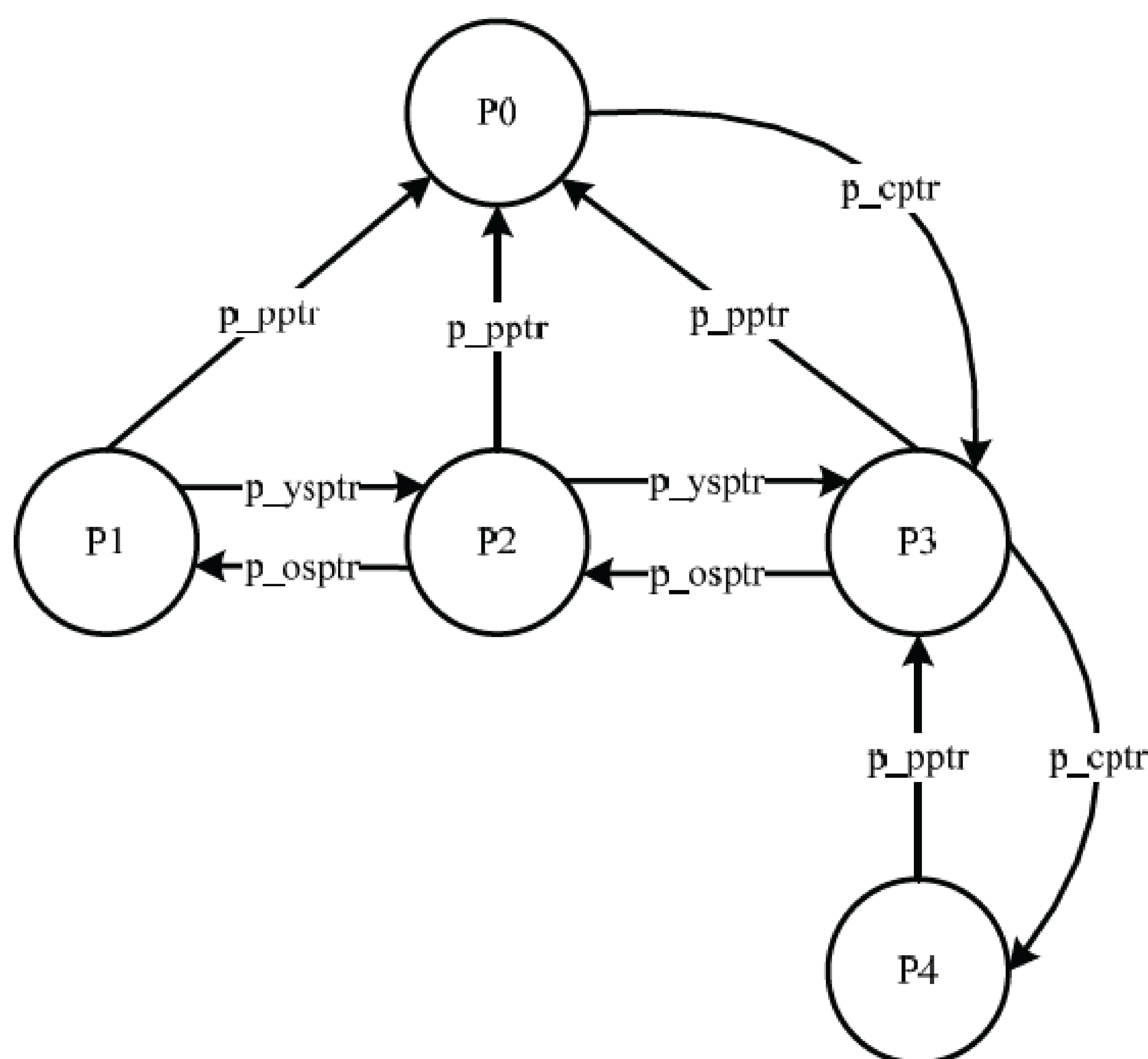


图 7.1　进程之间的关系

- p_opptr（祖先，original parent）：其指向创建进程 P 的进程描述符，如果父进程不存在，则指向进程 init 的描述符，所以当一个 Shell 用户启动一个后台进程并从 Shell 退出的时候，后台进程将变成 init 的子进程。

- p_pptr（父进程，parent）：其指向进程的父进程，其值通常来说和 p_opptr 一致，但是也可能不同。
- p_cptr（子进程，child）：指向进程年龄最小的子进程的描述符，即进程上一次创建的进程描述符。
- p_ysptr（弟进程，younger sibling）：指向在本进程创建之后由父进程创建的进程。
- p_osptr（兄进程，older sibling）：指向在本进程创建之前由父进程创建的进程。

3. 进程的类型

Linux 操作系统通常包括三种不同类型的进程，每种进程都有自己的特点和属性。

- 交互进程：由一个 Shell 启动的进程，其既可以在前台运行，也可以在后台运行。
- 批处理进程：这种进程和终端没有联系，是一个进程序列。
- 守护进程：Linux 系统启动时启动的进程，并在后台运行。

4. 进程的状态

进程在其生存周期内可能处于以下状态中，需要注意的是这些状态是互斥的，也就是说在同一时刻进程只能位于其中一个状态，在 task_struct 结构的状态域中使用不同关键字来定义这些状态。

- 可运行状态（TASK_RUNNING）：占用处理器执行或者准备执行。
- 可中断的等待状态（TASK_INTERRUPTIBLE）：进程被挂起或者睡眠，当某些条件变成真的时候才退出这种等待状态，这些条件包括。硬件中断、进程正在等待的系统资源被释放、传递一个信号等，退出等待状态之后的进程会回到 TASK_RUNNING 状态。
- 不可中断的等待状态（TASK_UNINTERRUPTIBLE）：和上一个状态类似，其差别是当接收到信号的时候并不能退出这个等待状态。
- 暂停状态（TASK_STOPPING）：进程的执行被暂停，通常来说当进程接收到 SIGSTOP、SIGTTIN 或者 SIGTTOU 信号后，进入暂停状态。需要注意的是如果一个进程被另外一个进程监控的时候，任何信号都可以把这个进程置于 TASK_STOPPEN 状态。
- 僵尸状态（TASK_ZOMBIE）：进程的执行已经被终止，但是父进程还没有使用 wait 系列系统调用已返回的相应信息，此时内核不能丢弃包含在该进程中的相应数据，因为父进程还可能需要这些数据。

进程在这几种状态之间相互转化，但对于用户而言是透明的，这个切换的过程也常常被称为进程的调度。

进程是一个随执行过程不断变化的实体。和程序要包含指令和数据一样，进程也包含程序计数器和所有处理器寄存器的值，同时它的堆栈中存储着（如子程序）参数、返回地址以及变量之类的临时数据。当前的执行程序，或者说进程，包含着当前处理器中的活动状态。在多处理操作系统中，进程具有独立的权限与职责。如果系统中某个进程崩溃，不会影响到其余的进程。每个进程运行在各自的虚拟地址空间中，通过一定的通信机制，它们之间才能发生联系。

7.2.2　Linux 的进程标识方法

在 Linux 中有很多进程在同时运行，可以使用两种方式对这些进程进行标识：进程描述符的地址或者进程标识符；对于当前操作系统中的每个独立进程来说，其对应的进程描述符的地址以及进程标识符都是唯一的。

1. 进程描述符

为了对进程进行管理，Linux 内核必须了解每个进程当前的执行状态，这些状态包括进程的优先级、进程的运行状态、进程分配的地址空间等。为了达到这个目的，Linux 内核提供了一个 task_struct 类型的结构体进程描述符（process descriptor）来存放这些相关信息。

Linux 内核提供了一个数组 task 用于存放进程描述符，其包含指向系统中所有 task_struct 结构的指针。这意味着系统中的最大进程数目受 task 数组大小的限制，缺省值一般为 512。创建新进程时，Linux 将从系统内存中分配一个 task_struct 结构并将其加入 task 数组。当前运行进程的结构用 current 指针来指示。

以下是一个进程描述符的主要结构及其说明：

```
struct task_struct {
volatile long state;
//进程的运行时状态，-1 代表不可运行，0 代表可运行，>0 代表已停止。
unsigned int flags;
//flags 是进程当前的状态标识，具体说明如下：
//0x00000002 表示进程正在被创建
//0x00000004 表示进程正准备退出
//0x00000040  表示此进程被 fork 出，但是并没有执行 exec
//0x00000400 表示此进程由于其他进程发送相关信号而被杀死
 unsigned int rt_priority;
//进程的运行优先级
 truct list_head tasks;
 //list_head 结构体
 struct mm_struct *mm;
 //内存使用的相关情况
  int exit_state;
  int exit_code, exit_signal;
  pid_t pid;
 //进程标识号
  pid_t tgid;
 //进程组号
 struct task_struct *real_parent;
 //real_parent 是该进程的“亲生父亲”，不管其是否被“寄养”
  struct task_struct *parent;
 //parent 是该进程现在的父进程，有可能是“继父”
  struct list_head children;
 //children 指的是该进程孩子的链表，可以得到所有子进程的进程描述符
 struct list_head sibling;
 //sibling 是该进程兄弟的链表，也就是其父亲的所有孩子的链表，用法与 children 相似
struct task_struct *group_leader;
```

```
  //主线程的进程描述符
  struct list_head thread_group;
  //进程所有线程的链表
  处理器 time_t utime, stime;
//进程相关时间
  struct timespec start_time;
  struct timespec real_start_time;
  //进程启动时间
char comm[TASK_COMM_LEN];
  //这个是该进程所有线程的链表
    int link_count, total_link_count;
  //文件系统信息计数
    struct thread_struct thread;
  //特定处理器下的状态
    struct fs_struct *fs;
  //文件系统相关信息结构体
    struct files_struct *files;
  //打开的文件相关信息结构体
    struct signal_struct *signal;
    struct sighand_struct *sighand;
  //信号相关信息的句柄
    unsigned long timer_slack_ns;
    unsigned long default_timer_slack_ns;
  //松弛时间值，用来规定 select()和 poll()的超时时间，单位是纳秒
};
```

2. 进程标识符

进程标识符（Process ID）是进程描述符中最重要的组成部分，其是一个在当前 Linux 系统中唯一的非负整数，用于标识和对应唯一的进程。

Linux 内核使用了一个数据类型 pid_t 来存放进程的进程标识符，这个数据类型的实质是一个 32 位的无符号整型数据。进程标识符被顺序编号，通常来说是前一个进程的进程标识符的值加 1。进程标识符是可以重复使用的，当一个进程被回收之后，过一段时间，其标识符又可以被再次使用。为了和 16 位处理器架构的应用系统相兼容，在 Linux 内核上通常允许使用的进程标识符是 0~32767。

在 Linux 中，有如下几个特殊的进程标识符所对应的进程。

- 进程标识符 0：对应的是交换进程（swapper），其用于执行多进程的调用。
- 进程标识符 1：对应的是初始化进程（init），在自举过程结束时由内核调用，其对应的文件是/sbin/init，负责 Linux 的启动工作，这个进程在系统运行过程中是不会终止的，可以说当前操作系统中的所有进程都是这个进程衍生而来的。
- 进程标识符 2：可能对应页守护进程（pagedaemon），用于虚拟存储系统的分页操作。

使用命令 ps-aux 可以查看系统中当前正在运行的进程标识符以及其他一些信息，以下列出了开始的几个进程：

```
USER     PID  %处理器  %MEM   VSZ   RSS  TTY  STAT START  TIME   COMMAND
root     1    0.0   0.0   3632   1972 ?    Ss         Jul02  0:00   /sbin/init
root     2    0.0   0.0   0      0 ?       S          Jul02  0:00   [kthreadd]
root     3    0.0   0.0   0      0 ?       S          Jul02  0:05   [ksoftirqd/0]
root     6    0.0   0.0   0      0 ?       S          Jul02  0:00   [migration/0]
root     7    0.0   0.0   0      0 ?       S          Jul02  0:02   [watchdog/0]
root     8    0.0   0.0   0      0 ?       S          Jul02  0:00   [migration/1]
root     10   0.0   0.0   0      0 ?       S          Jul02  0:05   [ksoftirqd/1]
```

可以使用 getpid 系列函数来获得当前进程的进程标识符，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

getpid 函数用于获得当前调用进程的进程标识符，getppid 用于获得当前调用进程的父进程的进程标识符，这两个函数的返回值都是对应的进程标识符。

3. Linux 进程的用户

和第 4.4.3 小节中介绍的 Linux 下的文件访问权限类似，进程也有对应的实际用户 ID、实际组 ID、有效用户 ID、有效组 ID，对于这些用户而言每个进程同样存在一个相应的标识符，Linux 提供了相应的函数用于获取这些标识符，对其标准调用格式说明如下：

```
#include <unistd.h>
#include <sys/types.h>
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

对以上 4 个函数的说明如下。

- getuid：返回进程的实际用户标志符。
- geteuid：返回调用进程的有效用户标识符。
- getgid：返回调用进程的实际组标识符。
- getegid：返回调用进程的有效组标识符。

4. 进程标识的获取实例

【例 7.1】使用 getgid 系列函数获取当前进程的相关标识符

例 7.1 是一个在 Linux 中获取当前进程的相关标识符实例，应用代码分别调用了 getpid、getppid、getuid、geteuid、getgid 和 getegid 函数来获取对应的标识符，并且通过 printf 函数将这些标识符依次输出，最后退出，其流程如图 7.2 所示。

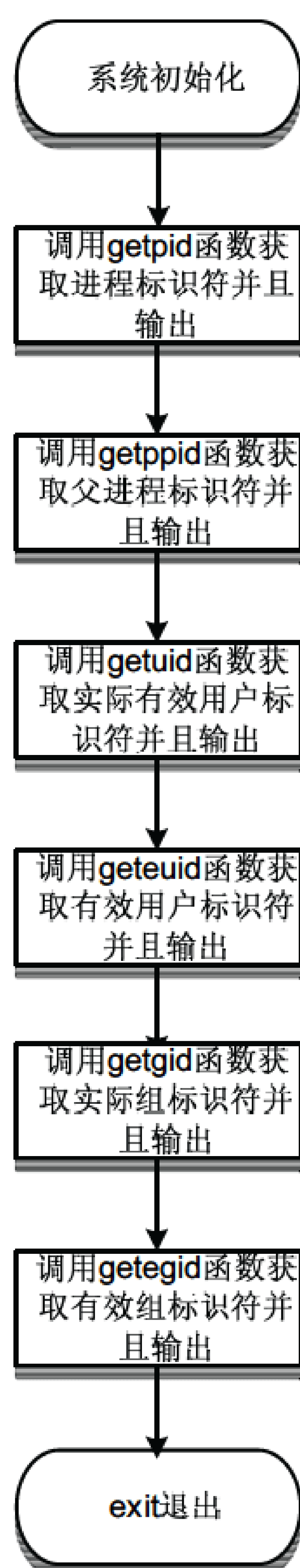


图 7.2　获取当前进程的相关标识符

实例的应用代码如下：

```
1    /*这是一个使用 getpid 等函数来获得当前进程的标识符等信息的实例
2    实例调用 printf 函数分别输出当前进程的标识符、父进程的标识符、
3    实际用户的标识符、有效用户的标识符、实际组标识符和有效组标识符*/
4    #include <sys/types.h>
5    #include <unistd.h>
6    #include <stdio.h>
7    int main(int argc,char *argv[])
8    {
9       printf("当前进程标识符是%d\n",getpid());             //进程标识符
10      printf("当前父进程标识符是%d\n",getppid());          //父进程标识符
11      printf("当前实际用户标识符是%d\n",getuid());         //实际用户标识符
12      printf("当前有效用户标识符是%d\n",geteuid());        //有效用户标识符
13      printf("当前实际组标识符是%d\n",getgid());           //实际组标识符
14      printf("当前有效组标识符是%d\n",getegid());          //有效组标识符
15      return 0;
16   }
```

将文件保存为 exam701getpid.c，在终端中使用 gcc 编译链接，生成可执行文件 exam701getpid.c

```
alloy@ubuntu:~/linuxc$ gcc exam701getpid.c -o exam701getpid
```

在当前目录下运行 exam701getpid 可执行文件，可以看到输出当前进程对应的各个标识符。

```
alloy@ubuntu:~/linuxc$ ./exam701getpid
当前进程标识符是 2740
当前父进程标识符是 2534
当前实际用户标识符是 1000
当前有效用户标识符是 1000
当前实际组标识符是 1000
当前有效组标识符是 1000
```

7.2.3 Linux 的进程调度

在 Linux 系统中，进程有两种运行模式：用户模式和系统模式。用户模式的权限比系统模式下的小很多，对于一般的进程，都是部分时间运行于用户模式，部分时间运行于系统模式。进程通过系统调用在这两种模式之间切换；当系统调用发生时，进程将由用户模式切换到系统模式继续执行；当系统调用返回时，进程将由系统模式切换回用户模式。

在 Linux 系统中，进程不能被抢占，只要能够运行它们就不会被停止。当进程必须等待某个系统事件时，它才决定释放出处理器。进程常因为执行系统调用而需要等待。由于处于等待状态的进程还可能占用处理器时间，所以 Linux 采用了预加载调度策略。在此策略中，每个进程只允许运行很短的时间（200ms），当这个时间用完之后，系统将选择另一个进程来运行，原来的进程必须等待一段时间以继续运行，这段时间称为时间片。

可运行进程是一个只等待处理器资源的进程。Linux 使用基于优先级的简单调度算法来选择下一个运行进程。当选定新进程后，系统必须将当前进程的状态、处理器中的寄存器以及上下文状态保存到 task_struct 结构中。同时它将重新设置新进程的状态并将系统控制权交给此进程。为了将处理器时间合理地分配给系统中每个可执行进程，调度管理器必须将这些时间信息也保存在 task_struct 中。

在 task_struct 结构中保存的调度信息如表 7.1 所示。

表 7.1　进程调度信息

字段名	含义
policy	该字段表示了进程的调度策略。系统中有两类进程：普通与实时进程。实时进程的优先级要高于普通进程，实时进程也有两种策略：时间片轮转和先进先出。
priority	该字段表示了实时进程的相对优先级
rt_priority	该字段表示了实时进程的相对优先级
counter	该字段表示了进程允许运行的时间。进程首次运行时为进程优先级的数值，它随时间的变化而递减

7.2.4 Linux 下的进程执行流程

图 7.3 是 Linux 下一个标准进程从开始建立到取消的详细过程，本章的下一个小节将按照这个步骤详细介绍 Linux 下的进程操作相关知识。

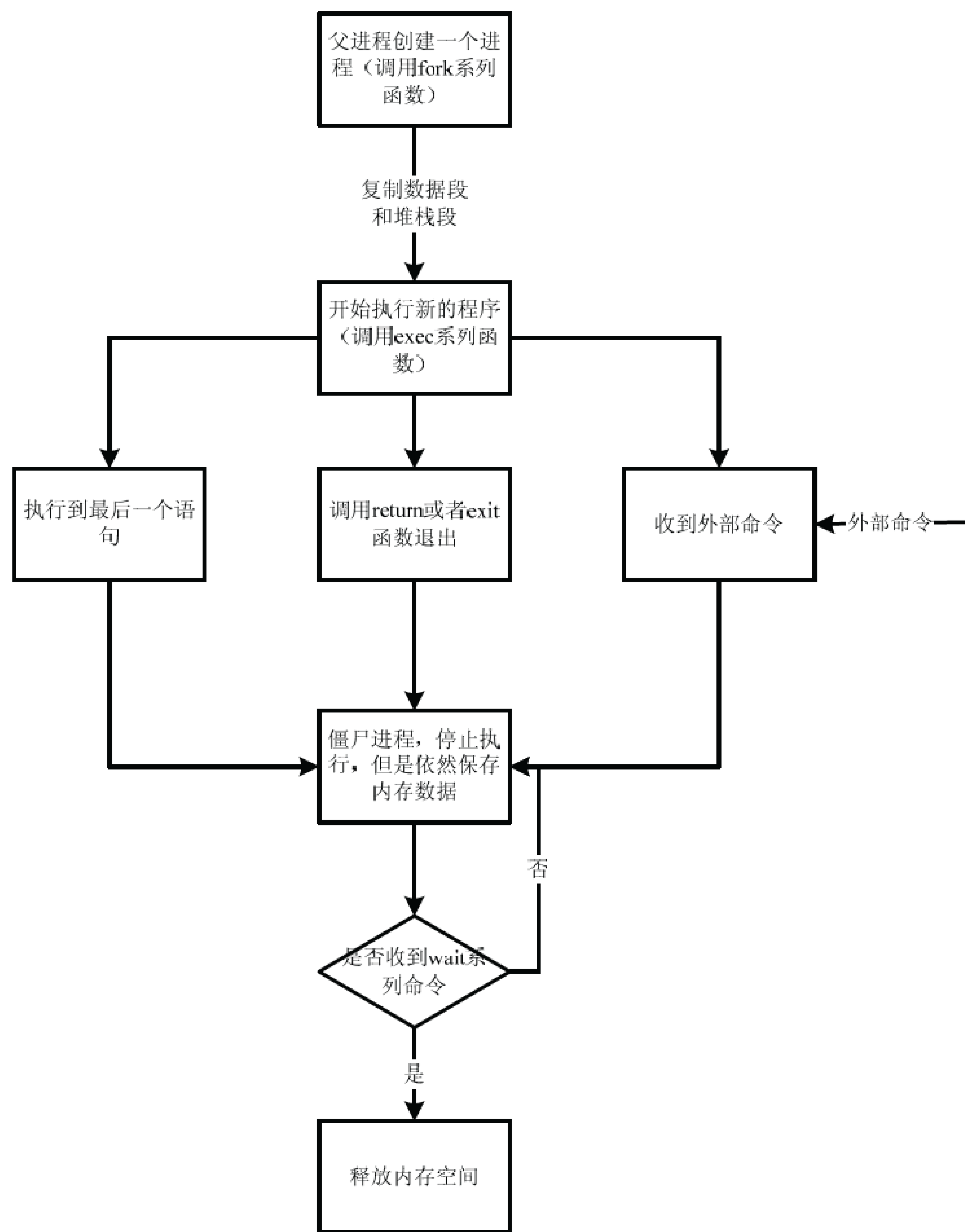


图 7.3 Linux 下的进程执行过程

7.3 Linux 的进程操作

Linux 的进程操作通常包括创建、执行、退出和销毁共 4 个步骤，如图 7.3 所示，Linux 提供了相应的函数对这些步骤进行操作。

7.3.1 使用 fork 函数来创建进程

在 Linux 中，创建一个新进程的唯一方法是由某个已存在的进程调用 fork 或 vfork 函数（将在第 7.3.3 小节中进行介绍），被创建的新进程称为子进程（child process），已存在的进程称为父进

程（father process）。

1. fork 函数基础

fork 函数的实质是一个系统调用（和 write 函数类似），其作用是创建一个新的进程，当一个进程调用它，完成后就出现两个几乎一模一样的进程，其中由 fork 创建的新进程被称为子进程，而将原来的进程称为父进程。子进程是父进程的一个拷贝，即子进程从父进程得到了数据段和堆栈段的拷贝，这些需要分配新的内存；而对于只读的代码段，通常使用共享内存的方式访问。

用户通常在有如下需求的时候使用 fork 函数：

- 一个进程希望复制自身，从而使得父子进程能同时执行不同段的代码，通常来说这种应用会涉及网络服务：父进程等待远端的一个请求或者应答，当收到这个请求或者应答的时候调用 fork 创建一个子进程来完成处理，而自己继续等待远端的请求或者应答。
- 进程想执行另外一个程序，例如在 Shell 中调用用户所生成的应用程序。

对 fork 函数的标准调用格式说明如下：

```
#include <unistd.h>
pid_t fork(void);
```

fork 函数没有参数，其被调用一次，但是返回两次：

- 对于父进程而言：函数的返回值是子进程的进程标识符，因为一个进程的子进程可以多于一个，所以没有一个函数使一个进程可以获得其所有子进程的进程标识符，必须通过这种方式来收集。
- 对于子进程而言：函数的返回值是 0，一个进程只会有一个父进程，所以子进程总是可以调用 getppid 以获得其父进程的进程标识符，所以不需要在这里返回父进程的进程标识符。
- 如果出错：返回值为“-1”。

所以用户可以通过 fork 函数的返回值来分辨父进程和子进程，例 7.2 是一个使用 fork 函数来创建子进程的实例。

【例 7.2】使用 fork 函数创建进程

应用代码调用 fork 函数创建了一个子进程，然后通过对其返回值分辨父进程和子进程，并且分别输出一个字符串，其流程如图 7.4 所示。

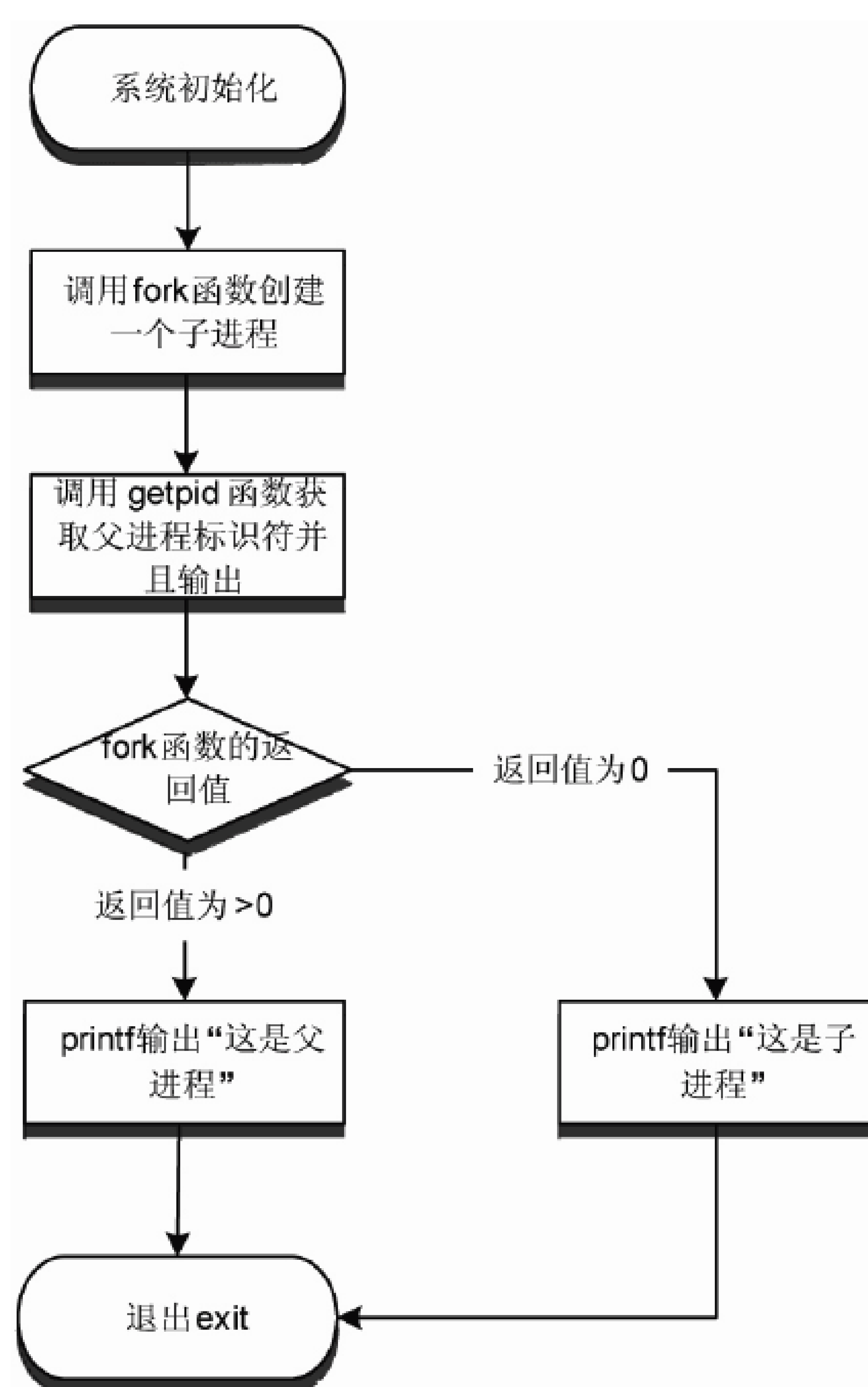


图 7.4　使用 fork 创建子进程

实例的应用代码如下：

```
1  /*这是一个调用 fork 函数创建子进程的实例，当创建进程成功之后会分别
2  打印两者对应的进程标识符*/
3  #include <stdio.h>
4  #include <stdlib.h>
5  int main(int argc,char *argv[])
6  {
7      pid_t pid;                      //进程标识符
8      pid = fork();                   //创建一个新的进程
9      if(pid < 0)                     //如果返回的 pid 小于 0，则标识创建进程失败
10         {
11             printf("创建进程失败!");
12             exit(1);                //fork 出错，退出
13         }
14     else if(pid == 0)               //如果 pid 为 0 则表示当前执行的是子进程
15         printf("这是子进程，进程标识符是%d\n",getpid());
16     else                            //否则为父进程
17         printf("这是父进程，进程标识符是%d\n",getpid());
18     return 0;                       //返回
19 }
```

将文件保存为 exam702fork.c，在终端中使用 gcc 编译链接生成可执行文件 exam702fork。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam702fork.c -o exam702fork
```

在当前工作目录下执行该可执行文件，可以看到如下的输出，其中父进程的进程标识符是

2757，而子进程的进程标识符是 2758。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam702fork
这是父进程，进程标识符是 2757
这是子进程，进程标识符是 2758
```

2. 子进程和父进程共享的数据空间

当 fork 函数返回后，子进程和父进程都从调用 fork 函数的下一条语句开始执行，但是父进程或子进程哪个先执行是随机的，这个取决于具体的调度算法，如果需要确定让其中一个先运行，可以使用 sleep 等函数让其中一个“休眠”一段时间，但是这个时间长度是不确定的。

通常来说，fork 所创建的子进程将会从父进程中拷贝父进程的数据空间、堆和堆栈，并且和父进程一起共享正文段，需要注意的是子进程所拷贝的仅仅是一个副本，和父进程的相应部分是完全独立的。

【例 7.3】在父进程和子进程中分别修改变量

例 7.3 是一个父进程和子进程分别对变量 var 和 glob 进行修改的应用实例，从其中可以看到子进程对变量的修改并不会影响到父进程中的变量，其流程如图 7.5 所示。

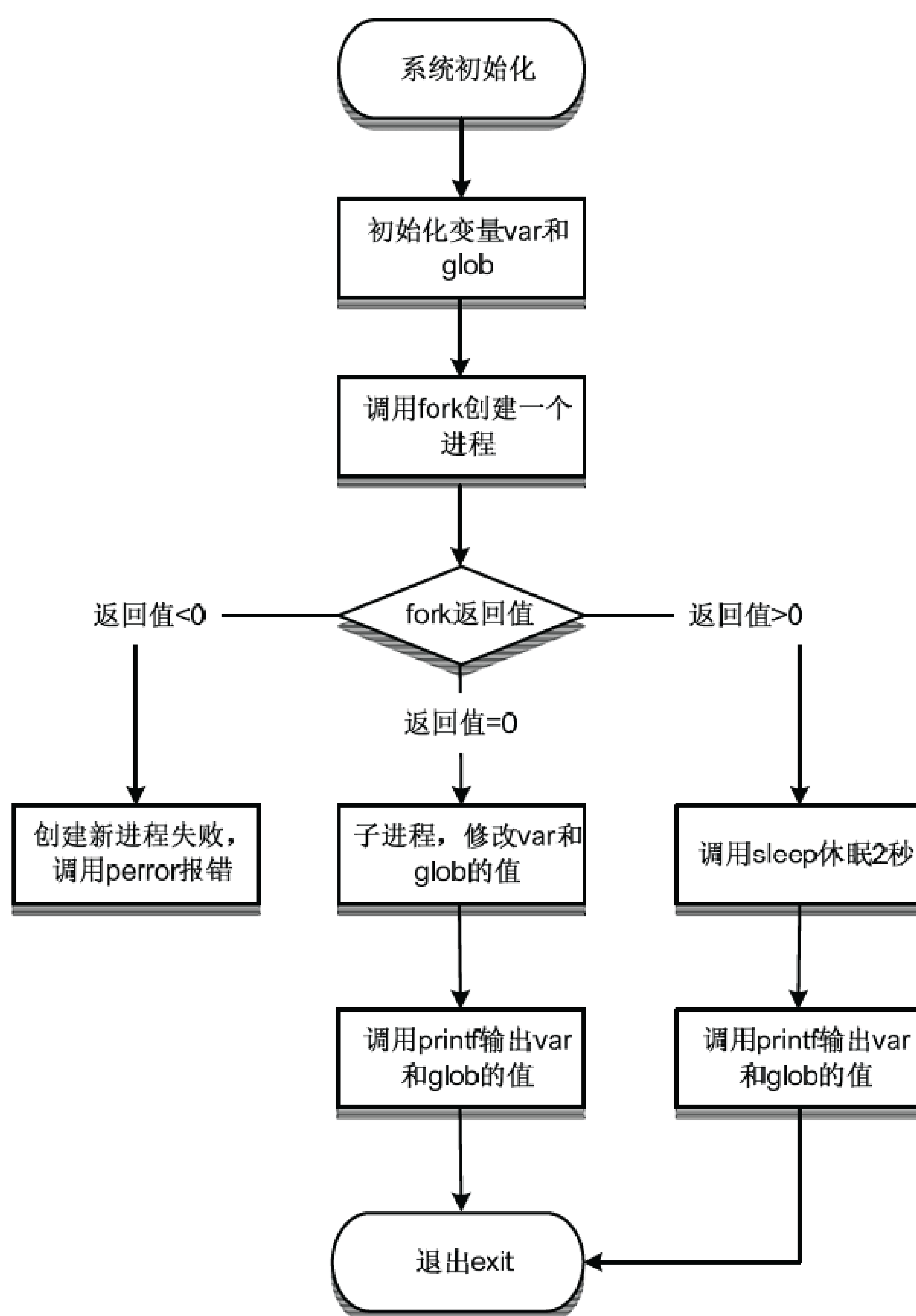


图 7.5　父进程和子进程分别对变量的修改

实例的应用代码如下：

```
1   /*这是一个调用 fork 函数创建一个子进程，然后分别打印输出子进程
2   和父进程中的变量的实例*/
3   #include <unistd.h>
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <errno.h>
7   int    glob = 6;                                   //外部变量
8   int main(int argc,char *argv[])
9   {
10          int       var;     //内部变量
11          pid_t     pid;     //文件标识符
12          var = 88;     //内部变量
13          printf("创建新进程之前。\n");                 //还没有创建子进程
14          if ((pid = fork()) < 0)                    //如果创建子进程失败
15          {
16             perror("创建子进程失败！");
17          }
18          else if (pid == 0)                         //现在是子进程
19          {
20             glob++;                                 //在子进程中修改变量值
21             var++;
22          }
23          else                                       //现在是父进程
24          {
25             sleep(2);                               //父进程阻塞 2 秒
26          }
27          printf("进程标识符为= %d, glob = %d, var = %d\n", getpid(), glob, var);
28          //分别在子进程输出两个变量的值
29          exit(0);
30  }
```

将文件保存为 exam703fork.c，在终端中调用 gcc 对其编译链接，生成可执行文件 exam703fork。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam703fork.c -o exam703fork
```

在当前工作目录下运行可执行文件 exam703fork，可以看到其中子进程的进程标识符为 2297，在其中 glob 和 var 变量已经被修改，而父进程中的进程标志符为 2296，其中的 glob 和 var 变量依然保持原来的值，需要注意的是行 27 的 printf 函数虽然只有一句，但是会分别在子进程和父进程中被执行一遍，所以有两个输出的字符串行，由于父进程调用了 sleep 语句对自身进行了休眠操作，所以其会比子进程晚两秒执行该 printf 函数语句。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam703fork
创建新进程之前。
进程标识符为= 2797, glob = 7, var = 89
进程标识符为= 2796, glob = 6, var = 88
```

3. 子进程和父进程共享的文件

上一小节介绍了在调用 fork 函数之后子进程将会复制父进程的相应内存空间，除此之外，父进程中所有打开的文件描述符也会被复制到子进程中，此时父进程和子进程的每个打开的文件描述符会共享同一个文件表项。

图 7.6 是父进程和子进程共享文件的示意图。

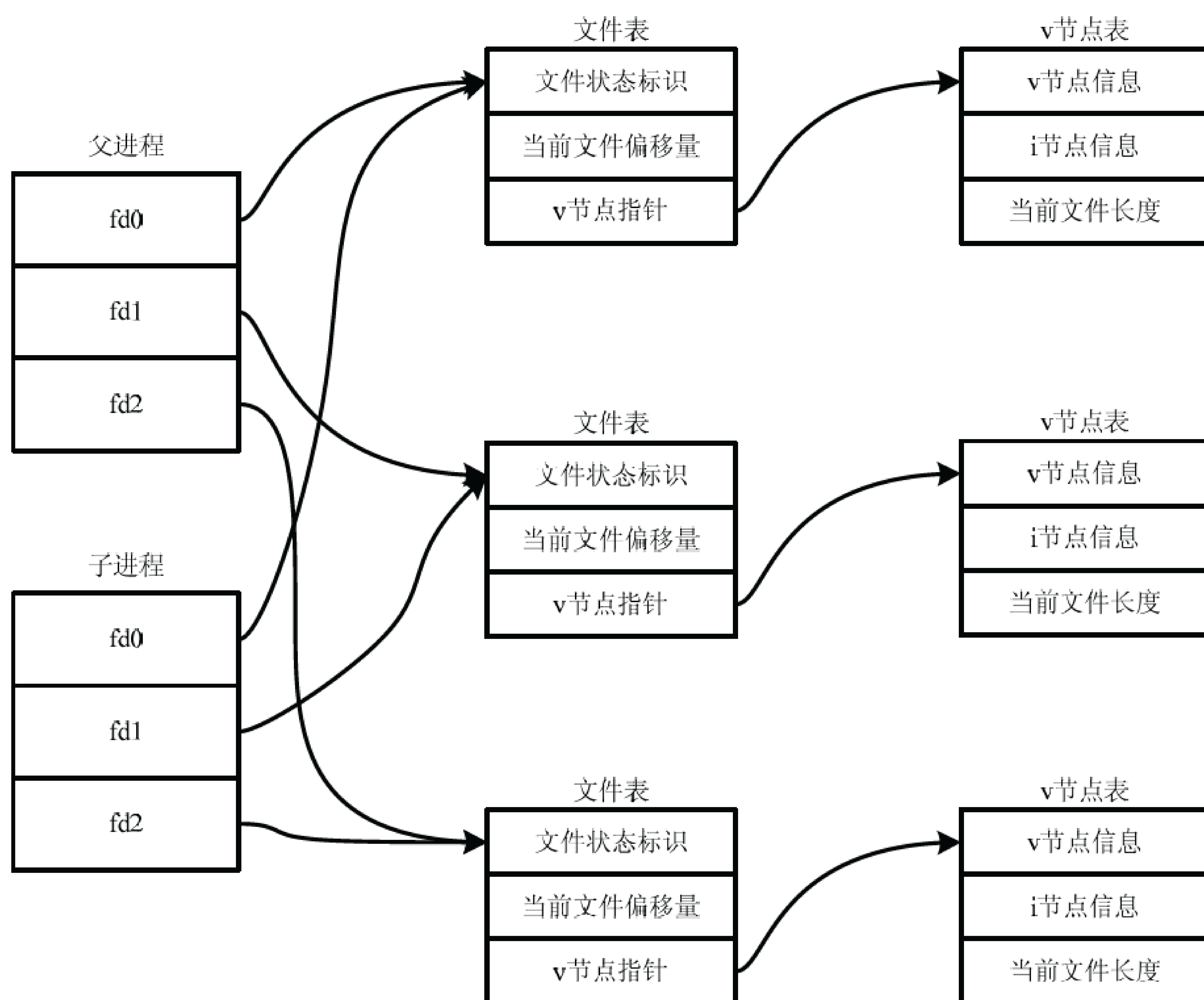


图 7.6 父进程和子进程对文件的共享

如图 7.6 所示，fork 所创建的子进程和父进程一起共享同一个文件的偏移量，此时如果父进程和子进程同时对同一个文件进行写操作且没有任何形式的同步操作，则会出现写文件的混乱，例 7.4 即为使用父进程和子进程同时写一个文件所导致的混乱实例。

【例 7.4】在父进程和子进程中分别对文件进行操作

应用代码将参数 argv[1]中指定文件中的类读出，然后写入到 argv[2]中所指定的文件中去，如果该文件不存在，则创建，其流程如图 7.7 所示。

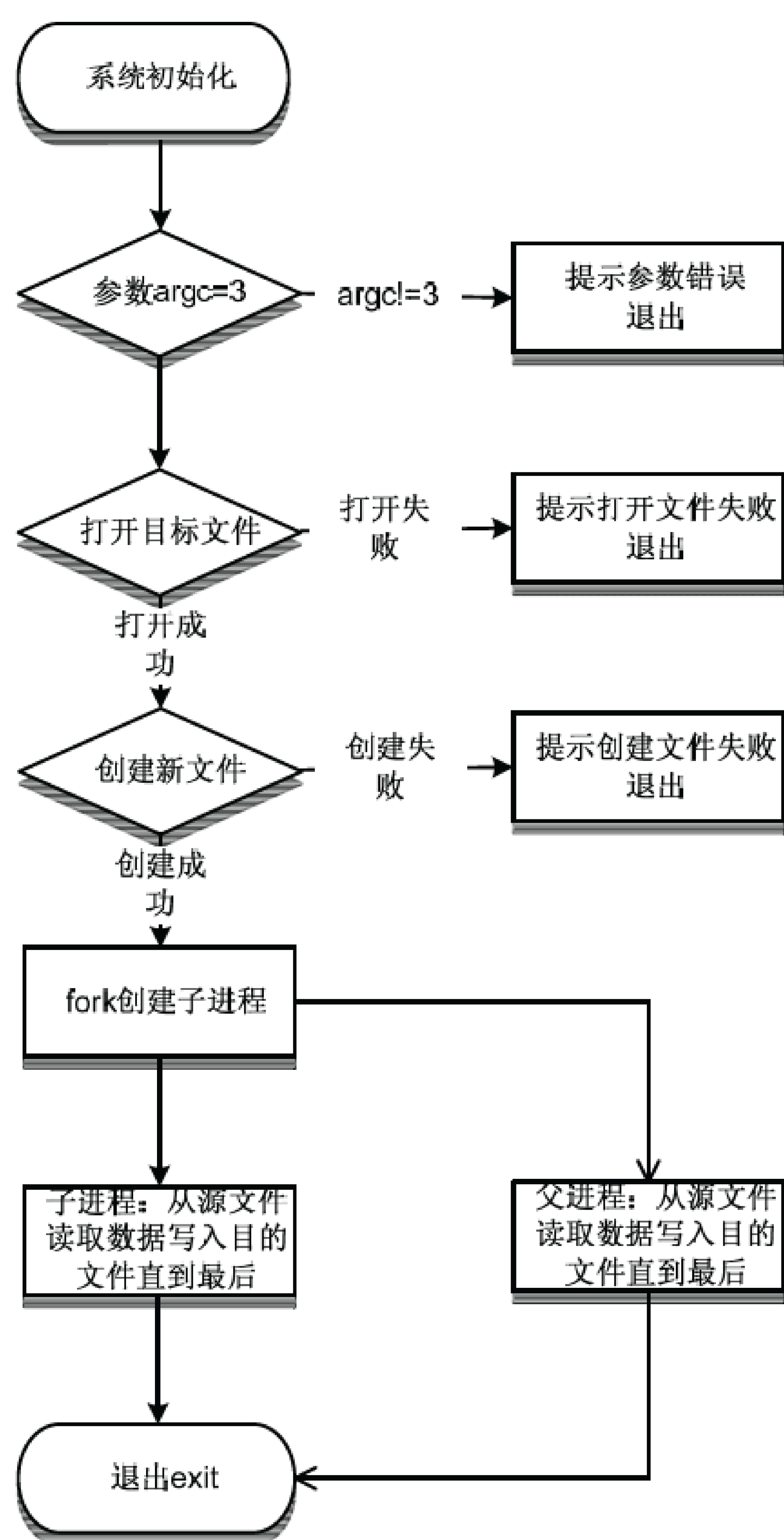


图 7.7　父进程和子进程同时对文件进行操作

实例的应用代码如下：

```
1   /*子进程和父进程同时对一个文件进行写操作导致文件发生混乱的实例
2   写入数据的文件由 argv[2]指定，数据来源在 argv[1]所指定的文件中*/
3   #include <sys/types.h>
4   #include <unistd.h>
5   #include <fcntl.h>
6   #include <stdio.h>
7   int readfd, writefd;                                    //读文件描述符和写文件描述符
8   char c;                                                 //文件内容的中转字符空间
9   int main(int argc, char*argv[])
10  {
11    if(argc!=3)                                           //如果参数不正确
12    {
13      printf("Usage %s sourcesfiel destfile. \n",argv[0]);
14      return 1;
15    }
16    if((readfd = open(argv[1], O_RDONLY))==-1)            //如果打开文件失败
```

```
17          {
18              printf("打开文件%s 失败！ \n",argv[1]);
19              return 2;
20          }
21          if((writefd = create(argv[2],S_IRWXU))==-1)              //如果创建文件失败
22      {
23              printf("创建文件%s 失败！ \n",argv[2]);
24              return 3;
25      }
26      fork();                  //创建子进程，以下为父进程和子进程同时执行的步骤
27      for(;;)
28      {
29          if(read(readfd,&c,1) != 1)                          //如果读不出数据则返回
30          {
31              return 4;
32          }
33          write(writefd,&c,1);                                //将读出的数据写入文件中
34      }
35      return 0;
36  }
```

将文件命名为 exam704forkfile.c，并且在 gcc 中进行编译，生成可执行文件 exam704forkfile。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam704forkfile.c -o exam704forkfile
```

在当前目录下建立一个名为 forkfiletest.txt 的文件，利用 vim 编辑其内容如下：

```
Sat Jul 27 12:32:57 2013
Sat Jul 27 12:32:58 2013
Sat Jul 27 12:32:59 2013
```

运行 exam704forkfile，其中第 2 个参数为一个新建立的待写入数据文件，使用 test.txt 作为其文件名，第 1 个参数为源数据文件，使用 forkfiletest.txt，此时 exam704forkfile 会使用子进程和父进程同时从 forkfiletest.txt 文件中读出数据，并且将其写入目标文件 test.txt。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam704forkfile forkfiletest.txt test.txt
```

使用“cat –n”命令查看 test.txt 文件的内容，可以看到如下输出：

```
alloy@ubuntu:~/linuxc/chapter7$ cat test.txt -n
  1   Sat Jul 27 12:32:5221
  2   a u 71:25
```

分别使用 test1.txt 和 test2.txt 作为目标文件的文件名，再次调用 exam704forkfile 进行读写操作，然后使用“cat –n”命令查看这两个文件的内容，会发现这两个文件的内容都不相同，这是因为子进程和父进程没有同步造成的。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam704forkfile forkfiletest.txt test1.txt
alloy@ubuntu:~/linuxc/chapter7$ cat test1.txt -n
     1  StJl2 23:721
     2  a u 71:25 03StJl2 23:921
alloy@ubuntu:~/linuxc/chapter7$ ./exam704forkfile forkfiletest.txt test2.txt
alloy@ubuntu:~/linuxc/chapter7$ cat test2.txt -n
1  Sat Jul 27 12:32:57 201
2  a u 71:25 03StJl2 23:921
```

从以上实例中可以看到，对于子进程和父进程来说，其运行的先后次序是不同的，并且其会共享一些资源，所以在实际使用中需要严格注意它们的同步，否则就会出现问题，通常来说可以在子进程或者父进程中使用 sleep 等语句对其进行阻塞以便确定先后执行顺序，以下是一个修改例 7.4 的应用代码，使用 sleep 语句将子进程休眠 2 秒来协调子进程和父进程工作的代码。

```
1   #include <sys/types.h>
2   #include <unistd.h>
3   #include <fcntl.h>
4   #include <stdio.h>
5   int readfd, writefd;                    //读文件描述符和写文件描述符
6   char c;                         //文件内容的中转字符空间
7   int main(int argc, char*argv[])
8   {
9     pid_t pid;
10    if(argc!=3)                   //如果参数不正确
11    {
12      printf("Usage %s sourcesfiel destfile. \n",argv[0]);
13      return 1;
14    }
15    if((readfd = open(argv[1], O_RDONLY))==-1)              //如果打开文件失败
16    {
17      printf("打开文件%s 失败！ \n",argv[1]);
18      return 2;
19    }
20    if((writefd = create(argv[2],S_IRWXU))==-1)             //如果创建文件失败
21   {
22      printf("创建文件%s 失败！ \n",argv[2]);
23      return 3;
24   }
25   pid = fork();          //创建子进程，以下为父进程和子进程同时执行的步骤
26   if(pid == 0)                                             //让子进程休眠 2 秒
27   {
28     sleep(1);
29   }
30   for(;;)
31   {
```

```
32          if(read(readfd,&c,1) != 1)                    //如果读不出数据则返回
33          {
34             return 4;
35          }
36          write(writefd,&c,1);                          //将读出的数据写入文件中
37       }
38       return 0;
39    }
```

将以上代码保存为文件 exam704forkfileOK.c，编译并且运行，使用 test3.txt 作为目的文件，然后使用“cat –n”命令查看文件内容，此时可以看到 test3.txt 文件中的内容是完全正常的。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam704forkfileOK.c -o exam704forkfileOK
alloy@ubuntu:~/linuxc/chapter7$ ./exam704forkfileOK forkfiletest.txt test3.txt
alloy@ubuntu:~/linuxc/chapter7$ cat test3.txt -n
 1    Sat Jul 27 12:32:57 2013
 2    Sat Jul 27 12:32:58 2013
 3    Sat Jul 27 12:32:59 2013
```

4. 创建多个子进程

在 Linux 中可以使用 fork 函数来创建多个子进程，以下即为一段使用 while 循环调用 fork 函数无限制创建子进程的代码，用户可以自行判断并且测试这个实例会产生什么样的后果。

```
1    #include <unistd.h>
2    int main(int argc,char *argv[])
3    {
4       while(1)
5       {
6          fork();
7       }
8    }
```



注　意

如果运行以上代码对应的可执行文件，由于无限制的创建进程导致硬件资源被分配光，Linux 会很快进入“死机状态”。

【例 7.5】使用 fork 创建多个子进程

例 7.5 是一个标准的创建两个子进程的实例，应用代码通过对 fork 函数返回值的判断来实现在父进程中继续创建第 2 个子进程，实例的流程如图 7.8 所示。

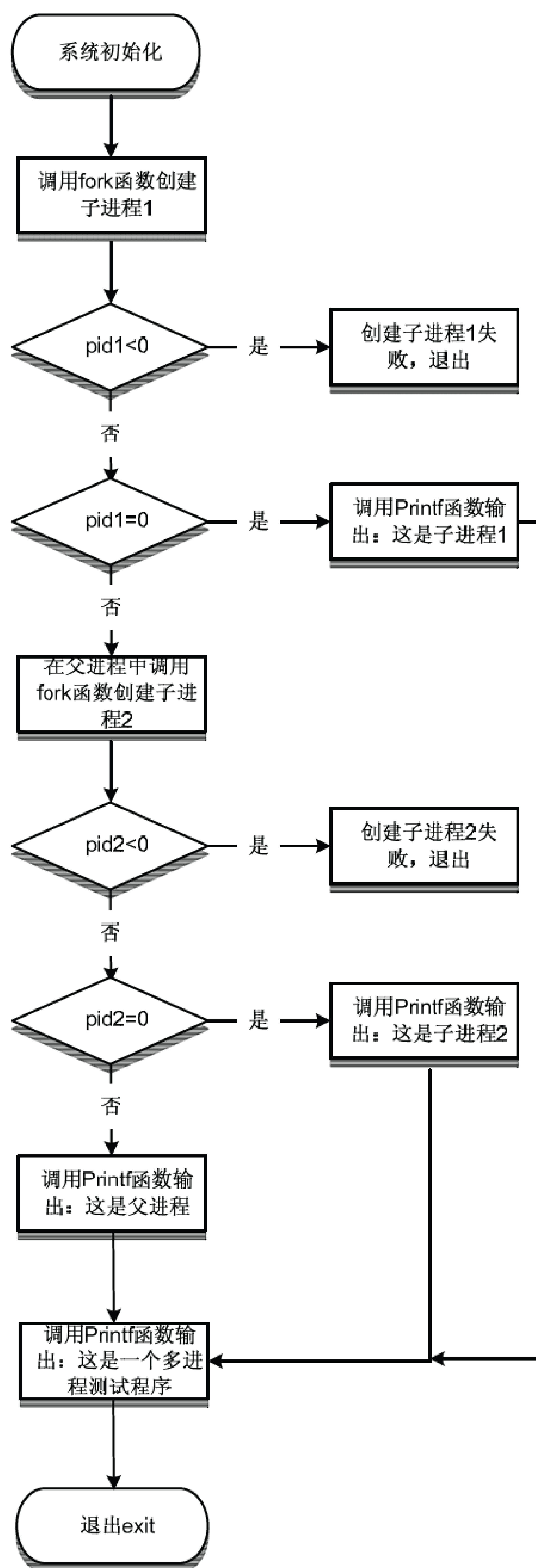


图 7.8　使用 fork 函数创建多个子进程

实例的应用代码如下：

```
1    /*这是一个调用 fork 函数创建子进程的实例，当创建进程成功之后会分别
2    打印两者对应的进程标识符*/
3    #include <stdio.h>
4    #include <stdlib.h>
```

```
5   int main(int argc,char *argv[])
6   {
7       pid_t pid1,pid2;                    //进程标识符
8       pid1 = fork();                      //创建一个新的进程
9       if(pid1 < 0)                        //如果返回的 pid 小于 0，则标识创建进程失败
10          {
11              printf("创建进程失败!\n");
12              exit(1);            //fork 出错，退出
13          }
14      else if(pid1 == 0)                  //如果 pid 为 0，则表示当前执行的是子进程
15      {
16          printf("这是子进程 1,进程标识符是%d\n",getpid());
17      }
18      else                        //否则为父进程
19      {
20          printf("这是父进程，进程标识符是%d\n",getpid());
21          pid2 = fork();
22          if(pid2 < 0)
23          {
24              printf("创建第二个进程失败！ \n");
25              exit(2);
26          }
27          else if(pid2 == 0)          //第 2 个子进程
28          {
29              printf("这是子进程 2，进程标识符是%d\n",getpid());
30          }
31          else
32          {
33              printf("这是子进程 2 的父进程，进程标志是%d\n",getpid());
34          }
35      }
36      printf("这是一个多进程测试，即将退出!\n");
37      return 0;           //返回
38  }
```

将以上代码保存为文件 exam705forksome.c，在终端中进行编译链接，生成可执行文件，exam705forksome.c。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam705forksome.c -o exam705forksome
```

执行 exam705forksome 可执行文件，可以看到父进程的进程标识符为 2859，而两个子进程的标识符分别是 2860 和 2861。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam705forksome
这是父进程，进程标识符是 2859
这是子进程 1,进程标识符是 2860
这是子进程 2 的父进程，进程标志是 2859
这是一个多进程测试，即将退出!
这是子进程 2，进程标识符是 2861
```

```
这是一个多进程测试，即将退出!
这是一个多进程测试，即将退出!
```

7.3.2 执行进程

在 Linux 中可以调用 fork 函数来创建一个子进程，该子进程几乎复制了父进程的全部内容，但是如果需要在子进程中执行一些自定义的动作，则需要调用 exec 函数族。

当进程调用 exec 系列函数的时候，该进程执行的程序被立即替换为新的程序，而新程序则从 main 函数开始执行，并且立刻替换掉了当前进程的正文段、数据段、堆和堆栈，需要注意的是其进程标识符和进程描述符是不会改变的。

1. exec 函数族基础

exec 函数族提供了一个在进程中启动另一个程序执行的方法，其可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新的进程替换了。

在 Linux 中通常会在如下两种情况下调用 exec 函数族：

- 当进程认为自己不能再为系统和用户做出任何贡献时，就可以调用 exec 函数族中的任意一个函数让自己重生。
- 如果一个进程想执行另一个程序，那么它就可以调用 fork()函数新建一个进程，然后调用 exec 函数族中的任意一个函数，这样看起来就像通过执行应用程序而产生了一个新进程（这种情况非常普遍）。

对 exec 系列函数的标准调用格式说明如下：

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],char *const envp[]);
```

如果这 6 个函数调用成功则没有返回值，如果出错则返回“-1”，对其参数说明如下。

- 参数 pathname：指出一个可执行目标文件的路径名。
- 参数 filename：指出可执行目标文件的文件名。
- 参数 arg：作为约定，同 pathname 一样指出目标文件的路径名。
- 参数 argv：是一个字符指针数组，由它指出该目标程序使用的命令行参数表，按照约定第一个字符指针指向与 pathname 或 filename 相同的字符串，最后一个指针指向一个空字符串，其余的指向该程序执行时所带的命令行参数。
- 参数 envp：与 argv 一样也是一个字符指针数组，由它指出该目标程序执行时的进程环境，它也以一个空指针结束。

事实上，这 6 个函数都是 exec 系列函数经过包装的库函数，它们的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件。这里的可执行文件既可以是二进制文件，也可以是任何 Linux 下可执行的脚本文件，图 7.9 是这 6 个 exec 系列函数之间的关系。



图 7.9　exec 函数族之间的关系

对 exec 系列函数的主要区别说明如下。

- execl 函数、execv 函数、execle 函数、execve 函数使用 pathname 参数，取路径名作为参数；而 execlp 函数和 execvp 函数，取文件名作为参数。
- execl 函数、execle 函数、execlp 函数中的“l”字符表示为“list”，其要求将新程序的每个命令行参数说明为一个单独的、以空指针为结尾的参数表；execv 函数、execve 函数和 execvp 函数中的“v”字符表示为“vector”，其要求先构造一个指向各个参数的指针，然后将该数组地址作为其参数。

execl、execel 和 execlp 这三个函数用于表示命令行参数的一般方式是：

```
char *arg0, char *arg1,....char *argn,(char *)0
```

需要注意的是在命令行参数中使用了一个将常数 0 强制转换为空指针的字符指针来作为结尾，因为如果进行强制转换，其会被解释为整型参数，从而出现错误。

execle 和 execve 函数中的最后一个字符“e”表示可以向新的进程传递一个环境变量 envp，对其命令参数说明如下：

```
char *arg0, char *arg1,....char *argn,(char *)0,char *envp[]
```

环境变量指的是一组值，这组值从 Linux 用户登录后就一直存在，很多应用程序需要依靠它来确定系统的一些细节，最常见的环境变量是路径（PATH），其指明了应到哪里去搜索相应的应用程序，如/bin；另外 HOME 也是比较常见的环境变量，其指明了用户在系统中的个人目录；环境变量一般以字符串“XXX=xxx”的形式存在，XXX 表示变量名，xxx 表示变量的值，例 7.6 是一个使用第 6 章中介绍的 printf 函数在屏幕上输出当前系统的环境变量实例。

【例 7.6】输出当前系统的环境变量

应用代码使用 while 语句分别对 argv 和 envp 参数进行操作，将其全部通过 printf 函数打印输出。

实例的应用代码如下：

```
1   /*这是一个输出 envp 环境变量的实例*/
2   #include <stdio.h>
3   int main(int argc, char *argv[ ], char *envp[ ])
4   {
5       printf("这是参数 argc\n%d\n", argc);            //首先打印参数的数目
6       printf("这是参数 argv\n");                      //以下打印参数列表
7       while(*argv)                                    //如果不为空，则输出这些字符串
8       {
9           printf("%s\n", *(argv++));
10      }
11      printf("这是环境变量 envp\n");                  //以下是 envp 字符串参数
12      while(*envp)                                    //输出 envp 参数
13      {
14          printf("%s\n", *(envp++));
15      }
16       return 0;
17  }
```

将文件保存为 exam706printfenvp.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam706printfenvp。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam706printfenvp.c -o exam706printfenvp
```

在当前目录下执行 exam706printfenvp，可以看到参数 argc、argv 和环境变量 envp 分别被输出。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam706printfenvp
这是参数 argc
1
这是参数 argv
./exam706printfenvp
这是环境变量 envp
LC_PAPER=zh_CN.UTF-8
LC_ADDRESS=zh_CN.UTF-8
LC_MONETARY=zh_CN.UTF-8
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=1bf275267199810c821d315500000029-1393387392.607011-762082811
SSH_CLIENT=192.168.0.102 53836 22
LC_NUMERIC=zh_CN.UTF-8
SSH_TTY=/dev/pts/1
USER=alloy
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;
31;01:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;
31:*.taz=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;
31:*.gz=01;31:*.lz=01;31:*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;
31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.ace=01;31:*.zoo=01;
31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;
35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;
35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;
35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;
```

```
35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;
35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;
35:*.axv=01;35:*.anx=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.mid=00;
36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.axa=00;
36:*.oga=00;36:*.spx=00;36:*.xspf=00;36:
LC_TELEPHONE=zh_CN.UTF-8
MAIL=/var/mail/alloy
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/arm/4.3.3/bin
LC_IDENTIFICATION=zh_CN.UTF-8
PWD=/home/alloy/linuxc/chapter7
LANG=zh_CN.UTF-8
LC_MEASUREMENT=zh_CN.UTF-8
SHLVL=1
HOME=/home/alloy
LANGUAGE=zh_CN:en
LOGNAME=alloy
SSH_CONNECTION=192.168.0.102 53836 192.168.0.103 22
LESSOPEN=| /usr/bin/lesspipe %s
LESSCLOSE=/usr/bin/lesspipe %s %s
LC_TIME=zh_CN.UTF-8
LC_NAME=zh_CN.UTF-8
_=./exam706printfenvp
OLDPWD=/home/alloy/linuxc
```

2. exec 函数族的应用

表 7.2 是 exec 函数族之间的区别和比较，其中“●”表明该函数有这个参数。

表 7.2 exec 函数族的比较

函数名	pathname 参数	filename 参数	参数表	argv[]	environ 参数	envp[]
execl 函数	●		●		●	
execlp 函数		●	●		●	
execle 函数	●		●			●
execv 函数	●			●	●	
execvp 函数		●		●	●	
execve 函数	●			●		●

在 exec 系列函数族执行之后，不仅进程的描述符、标识符没有发生改变，该进程的如下特征也将保留：

- 进程标识符和父进程标识符。
- 实际用户 ID、实际组 ID。
- 附加组 ID。
- 进程组 ID。
- 会话 ID。

- 闹钟剩余时间。
- 控制终端。
- 当前工作目录。
- 根目录。
- 文件模式创建屏蔽字。
- 文件锁。
- 进程信号屏蔽。
- 未处理信号。
- 资源限制。
- tms_utime、tms_stime、tms_cutime 以及 tms_cstime。

例 7.7~例 7.10 是 exec 函数族中几个常用函数的基本使用方法的实例。

【例 7.7】使用 execl 函数调用 date 命令

execl 是 exec 函数族中最常用的函数，例 7.6 是一个使用其调用 date 命令输出当前时间和日期信息的实例，需要注意的是由于在调用 execl 之后 main 函数的进程已经被 execl 所启动的 date 命令所替换，所以在这个 main 函数中只能使用一个 execl 函数。

实例的应用代码如下：

```
1    /*调用 execl 执行一个命令，需要注意的是在同一个进程中只能有一个 execl*/
2    #include<unistd.h>
3    int main(int argc,char *argv[])
4    {
5        execl("/bin/date","/bin/date",(char*)0);    //使用 execl 函数调用 date 命令
6        return 0;
7    }
```

将文件保存为 exam707excel.c，在终端中使用 gcc 编译链接，生成 exam707excel 可执行文件：

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam707excel.c -o exam707excel
```

在当前工作目录中运行 exam707excel 可执行文件，可以看到当前的时间输出。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam707excel
**This is a test for exec series fun**
2014 年 02 月 26 日 星期三 18:16:07 CST
```

【例 7.8】使用 execlp 函数调用 ls 命令

例 7.7 是一个调用 execlp 执行 ls 查找命令的实例，其会从 PATH 环境变量所指定的目录中查找符合参数 file 的命令或者文件，找到之后即可开始执行，然后以 argv[0]参数作为该命令的执行参数，在本应用中是使用 ls 查找命令来查找 etc 文件夹下的 passwd 文件。在该实例中 execlp 函数和 execl 函数在应用上有所不同，前者使用了环境变量作为 ls 命令所在位置的参数，所以不需要如例 7.7 一样写明 date 命令所在的路径。

实例的应用代码如下：

```
/*调用 execlp 执行 ls 命令，然后查找 passwd 文件*/
#include<unistd.h>
int main(int argc,char *argv[])
{
    execlp("ls","ls","-al","/etc/passwd",(char * )0);
    //执行 bin 下的 ls 命令，查找 etc 下的 passwd 文件，参数为 al
    return 0;
}
```

将文件保存为 exam708execlp.c，在终端中使用 gcc 编译链接，生成可执行文件 exam708execlp。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam708execlp.c -o exam708execlp
```

在当前目录中运行 exam708execlp 可执行文件，可以看到 passwd 文件的详细信息，其本质就是在/etc/passwd 工作路径下使用“ls –al”命令。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam708execlp
-rw-r--r-- 1 root root 1813   1 月   7 10:24 /etc/passwd
```

【例 7.9】使用 execv 函数调用 ls 命令

例 7.9 是一个使用 execv 函数重新实现查询功能的实例，execv 支持将参数放到一个数组中，然后传递。

实例的应用代码如下：

```
1    #include<unistd.h>
2    int main(int argc,char *argv[])
3    {
4      char *arg[ ]={"ls","-al","/etc/passwd",(char*)0};        //将参数放到一个数组中，然后传递
5      execv("/bin/ls",arg);                                    //执行 ls 命令，参数由 argv 数组传递
6      return 0;
7    }
```

将文件保存为 exam709execv.c，在终端中编译链接，生成可执行文件 exam709execv。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam709execv.c -o exam709execv
```

在当前工作目录下运行该文件，可以看到如下的输出，其实现的功能和例 7.8 是完全相同的。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam709execv
-rw-r--r-- 1 root root 1813   1 月   7 10:24 /etc/passwd
```

以上介绍了 exec 函数族中常见函数的应用方法， exec 函数族通常还是会结合 fork 函数在新建的进程中使用。例 7.10 是一个使用 fork 函数建立一个子进程，然后分别在子进程和父进程中使用 execl 函数调用两个命令的实例。

【例 7.10】在父进程和子进程中分别使用 execl 函数

应用代码调用 fork 函数创建一个子进程，然后通过对 fork 函数返回的 pid 进行判断以区别父进程和子进程，然后在父进程和子进程中分别使用 excel 调用了两个“ls –al”命令查询/etc/passwd

文件以及例 7.2 中创建的 exam702fork.c 文件的具体属性，其工作流程如图 7.10 所示。

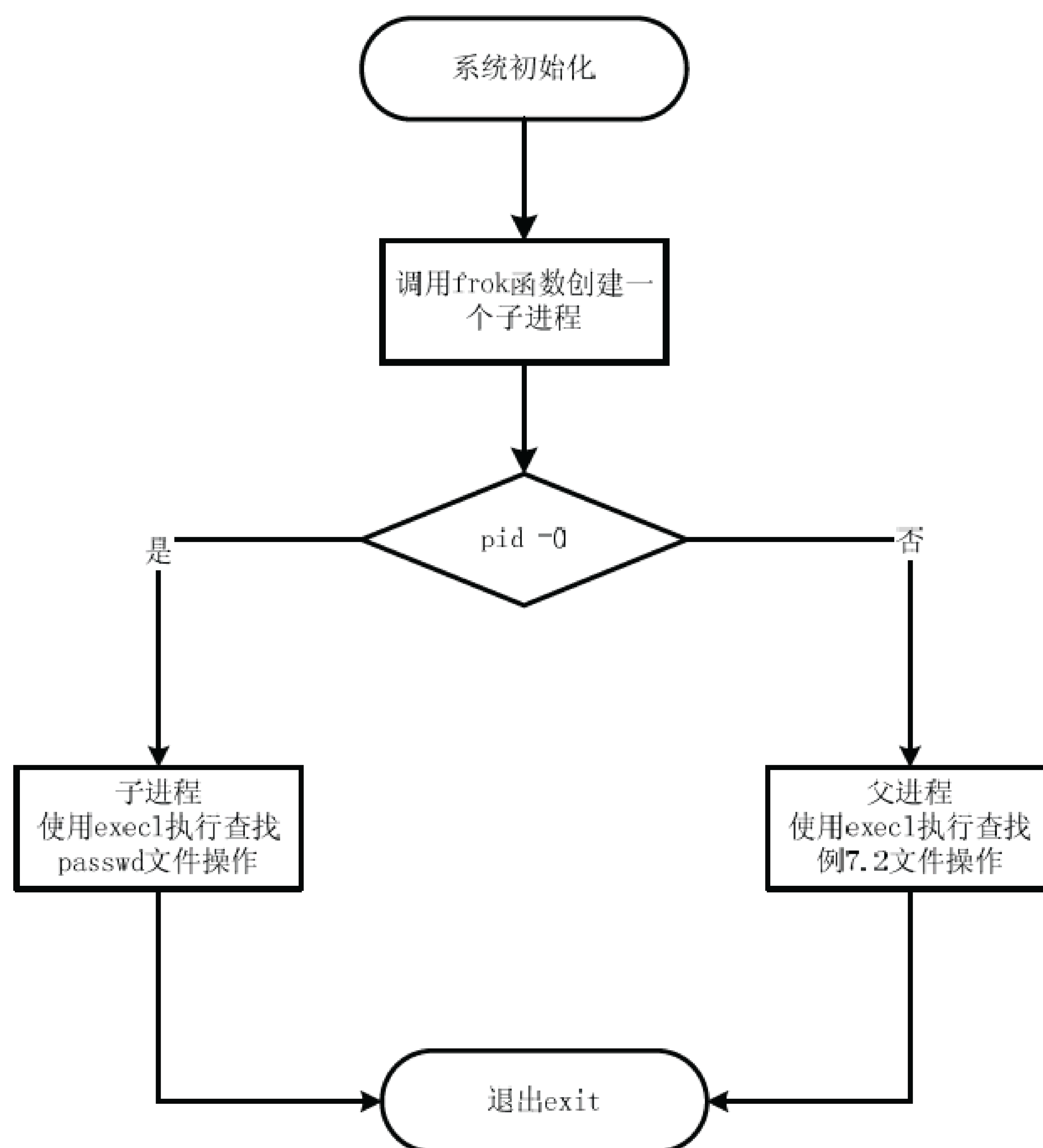


图 7.10　在父进程和子进程中分别使用 execl 函数

实例的应用代码如下：

```
1   #include<unistd.h>
2   #include<stdio.h>
3   #include<stdlib.h>
4   int main(int argc,char *argv[])
5   {
6       pid_t pid;
7       pid = fork();
8       if(pid == 0)     //子进程
9       {
10          execl("/bin/ls","ls","-al","/etc/passwd",(char * )0);
11      //执行 bin 下的 ls 命令，查找 etc 下的 passwd 文件，参数为 ls
12          exit(0);
13      }
14      else
15      {
16          execl("/bin/ls","ls","-al","./exam702fork.c",(char *)0);
17      //执行 bin 下的 ls 命令，查找当前文件夹下的 exam702fork.c 文件
18          exit(1);
19      }
20      return 0;
```

将文件保存为 exam710forkexec.c，在终端中使用 gcc 编译链接，生成可执行文件 exam710forkexec。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam710forkexec.c -o exam710forkexec
```

在当前工作路径下运行可执行文件，可以看到分别在子进程和父进程中输出了两个指定文件的属性。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam710forkexec
-rw-r--r-- 1 root root 1813    1 月    7 10:24 /etc/passwd
-rw-rw-r-- 1 alloy alloy 729    2 月  26 14:57 ./exam702fork.c
```

7.3.3 使用 vfork 函数创建并且执行进程

在使用 fork 函数创建一个新进程之后，可以不使用 exec 系列函数来执行新的程序，如果要执行新的程序则必须手动调用 exec 系列函数，在这种情况下可以使用 vfork 函数，vfork 函数在创建完一个新的进程之后自动实现 exec 系列函数的功能，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork(void);
```

函数的返回和 fork 函数类似，父进程中返回子进程的进程号，在子进程中返回 0，若出错则返回-1。

fork 与 vfork 之间的区别如下：

- fork 要拷贝父进程的数据段；而 vfork 则不需要完全拷贝父进程的数据段，在子进程没有调用 exec 系列函数或 exit 函数之前，子进程与父进程共享数据段。
- vfork 函数会自动调用 exec 系列函数去执行另外一个程序。
- fork 不对父子进程的执行次序进行任何限制；而在 vfork 调用中，子进程先运行，父进程挂起，直到子进程调用了 exec 系列函数或 exit 之后，父子进程的执行次序才不再有限制。

例 7.11 是一个使用 vfork 函数重写例 7.3 的实例。

【例 7.11】使用 vfork 创建子进程

应用代码使用 vfork 函数替代 fork 函数来创建子进程，由于 vfork 函数会自动让子进程先运行，所以不需要父进程调用 sleep 函数阻止自身运行。

实例的应用代码如下：

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <errno.h>
5   int   glob = 6;                          //外部变量
6   int   main(int argc,char *argv[])
```

```
7   {
8           int         var;                        //内部变量
9           pid_t       pid;                        //文件标识符
10          var = 88;                               //内部变量
11          printf("创建新进程之前。\n");            //还没有创建子进程
12          if ((pid = vfork()) < 0)                //如果创建子进程失败
13          {
14              perror("创建子进程失败！ ");
15          }
16          else if (pid == 0)                      //现在是子进程
17          {
18              glob++;                             //在子进程中修改变量值
19              var++;
20              //exit(0);
21          }
22          else                                    //现在是父进程
23          {
24               //glob = 101;
25               //var =   102;                   //修改变量的值
26               //sleep(2);                        //父进程阻塞 2 秒
27          }
28          printf("进程标识符为= %d, glob = %d, var = %d\n", getpid(), glob, var);
29          //分别在子进程中输出两个变量的值
30          exit(0);
31  }
```

将文件保存为 exam711vfork.c，在终端中使用 gcc 进行编译链接，生成 exam711vfork 可执行文件。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam711vfork.c -o exam711vfork
```

在当前工作目录中执行 exam711vfork，可以看到子进程对变量 glob 和 var 进行操作，改变了父进程中的变量值，这是因为 vfork 函数所创建的子进程是在父进程的内存空间中运行的。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam711vfork
创建新进程之前。
进程标识符为= 3075, glob = 7, var = 89
进程标识符为= 3074, glob = 7, var = 89
```

例 7.12 是另外一个 vfork 函数的应用实例。

【例 7.12】使用 vfork 创建子进程并且执行命令

应用代码分别利用子进程和父进程对一个 count 进行计数并且输出，用于展示父进程和子进程是共享一个数据段的，其流程如图 7.11 所示。

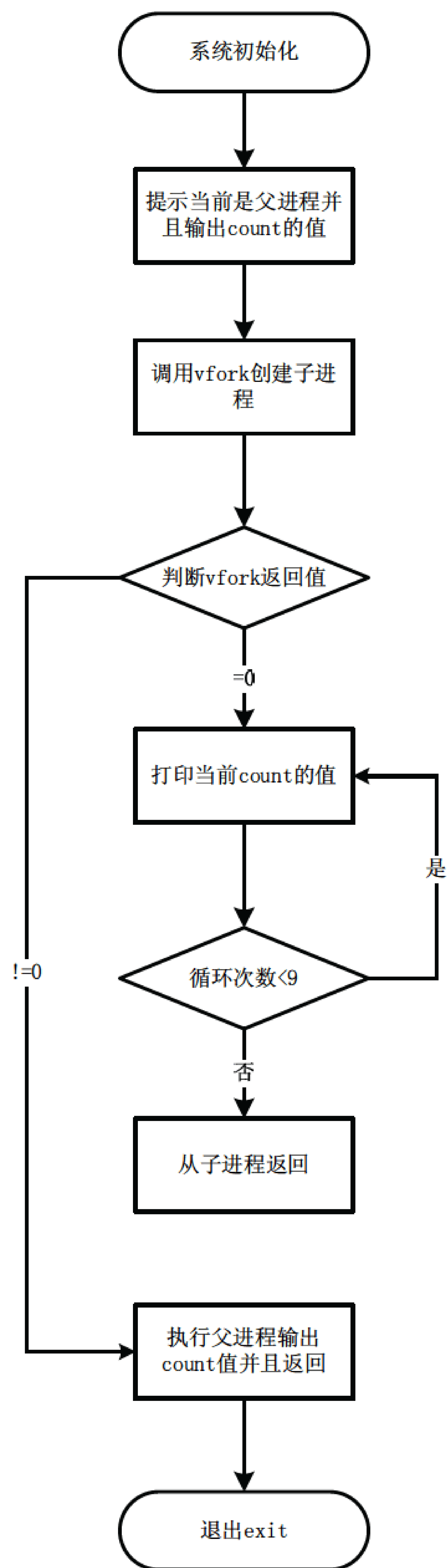


图 7.11　使用 vfork 创建子进程并且执行

实例的应用代码如下：

```
1  /*这是一个分别利用子进程和父进程对一个 count 进行计数并且输出，
2  用于展示父进程和子进程是共享一个数据段*/
3  #include <sys/types.h>
4  #include <unistd.h>
5  #include <stdio.h>
```

```
 6  #include <stdlib.h>
 7  int main (int argc,char *argv[])
 8  {
 9      int count = 1;
10      int child;
11      printf("Before create son, the father's count is:%d\n", count);       //创建子进程之前
12      if(!(child = vfork()))                                                 //创建子进程
13        {
14          //由于子进程会首先执行，以下为子进程执行过程
15          int i;
16          for(i = 0; i < 100; i++)
17            {
18              printf("这是子进程，当前 i 的值是: %d\n", i);                  //反复输出打印结果
19              if(i == 8)
20              exit(1);
21            }
22          printf("这是子进程，其进程 ID 是%d count 的值是: %d\n", getpid(), ++count);
23          exit(1);                                                           //退出子进程
24        }
25       else
26        {  //父进程执行区
27          printf("这是父进程，其进程 ID 是%d count 的值是: %d，其子进程是: %d\n",
            getpid(), count, child);
28        }
29  return 0;
30  }
```

将文件保存为 exam712vforksharedata.c，在终端中使用 gcc 编译链接，生成可执行文件 exam712vforksharedata。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam712vforksharedata.c -o exam712vforksharedata
```

执行该可执行文件，可以看到首先执行的是父进程，然后在子进程中更改计数器的值并在屏幕上输出。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam712vforksharedata
此时执行的是父进程，当前 count 的值是:1
这是子进程，当前 i 的值是: 0
这是子进程，当前 i 的值是: 1
这是子进程，当前 i 的值是: 2
这是子进程，当前 i 的值是: 3
这是子进程，当前 i 的值是: 4
这是子进程，当前 i 的值是: 5
这是子进程，当前 i 的值是: 6
这是子进程，当前 i 的值是: 7
这是子进程，当前 i 的值是: 8
这是父进程，其进程 ID 是 3130   count 的值是: 1，其子进程是: 3131
```

7.3.4　退出进程

在前面的实例中，调用 execl 函数族之后都使用了 exit 函数将进程退出，当一个进程执行完成之后必须要退出，退出时内核会进行一系列的相应操作，包括冲洗缓冲区等，在 Linux 中一共有 8

种进程的退出方法，其中包括 5 种正常的方法和 3 种异常退出。

通常来说 Linux 的应用代码会调用 exit 系列函数来退出一个进程，对其标准调用格式说明如下：

```
#include <stdlib.h>
#include <unistd.h>
void exit(int status);
void _exit(int status);
void _Exit(int status);
```

exit 系列函数没有返回值，其使用一个称为终止状态（exit status）的整型变量作为参数，Linux 内核会对这个终止状态进行检查；当异常终止时，Linux 内核会直接产生一个终止状态字，描述异常终止的原因，可以通过 wait 或者 waitpid 函数（将在下一小节进行介绍）来获得终止状态字；父进程也可以通过检查终止状态来获得子进程的状态。如果是以下三种状态：

- 在调用 exit 系列函数的时候不带终止状态。
- main 函数执行了一个无返回值的 return。
- main 函数的返回值不是一个整型。

则 Linux 会认为该进程的终止状态是未定义的，如果 main 函数的返回值定义为整型并且 main 函数是执行到最后一条语句返回，则该进程的终止状态是 0。



在 main 函数中调用 return 语句返回在绝大多数时等效于调用 exit 系列函数。

这两个函数的调用过程如图 7.12 所示。从图中可以看出：

- _exit 函数：直接使进程停止运行，清除其使用的内存空间，并清除其在内核中的各种数据结构。
- exit 函数：在_exit 的基础上做了一些包装，在执行退出之前加了若干道工序。

exit 函数与_exit 函数的最大区别在于：前者在调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件；而后者直接使进程停止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构，即图中的“清理 I/O 缓冲”一项。

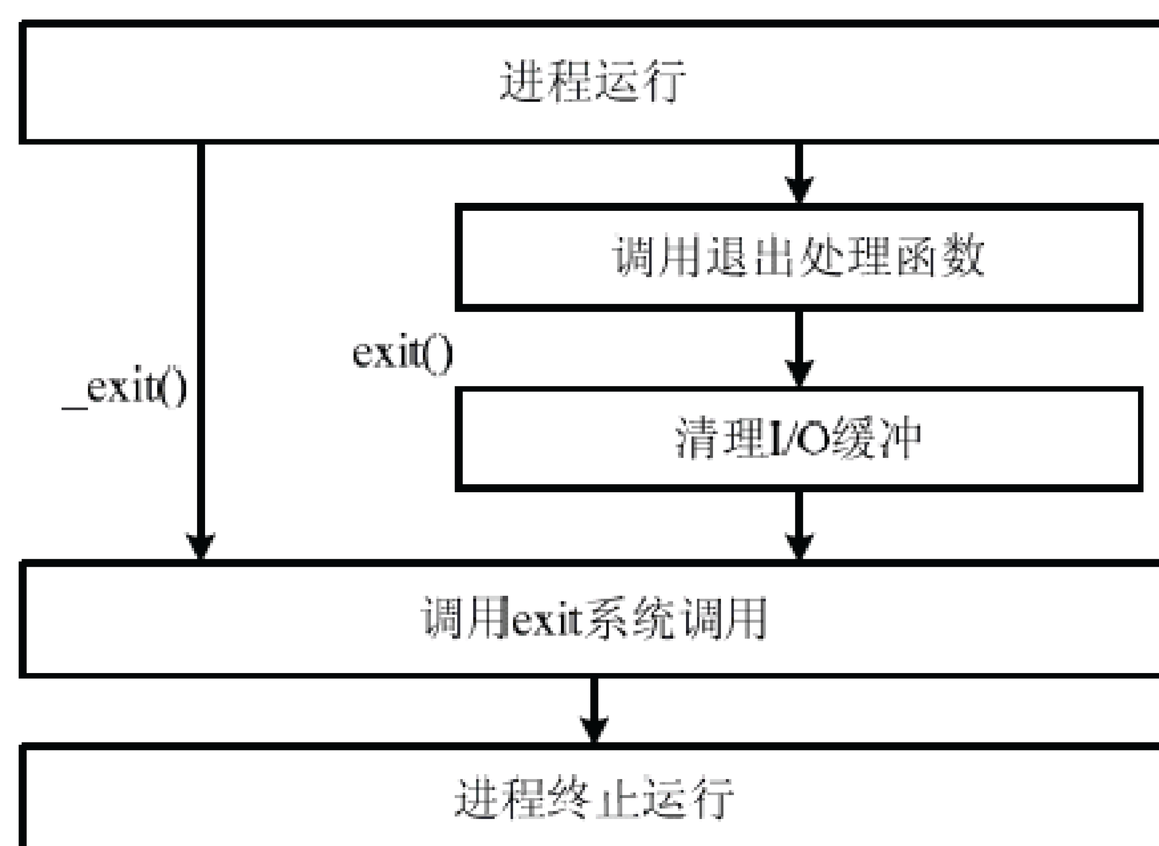


图 7.12　exit 函数和_exit 函数

由于在 Linux 的标准函数库中，有一种被称作“缓冲 I/O（buffered I/O）”操作，其特征就是对应每一个打开的文件，在内存中都有一片缓冲区。每次读文件时，会连续读出若干条记录，这样在下次读文件时就可以直接从内存的缓冲区中读取；同样，每次写文件的时候，也仅仅是写入内存中的缓冲区，等满足了一定的条件（如达到一定数量或遇到特定字符等），再将缓冲区中的内容一次性写入文件。

这种技术大大增加了文件读写的速度，但也为编程带来了一些麻烦。例如有些数据，认为已经被写入文件中，实际上因为没有满足特定的条件，它们还只是被保存在缓冲区内，这时用_exit 函数直接将进程关闭，缓冲区中的数据就会丢失，因此，若想保证数据的完整性，就一定要使用 exit 函数。

例 7.13 是一个利用 printf 函数利用只有读到换行符才会从缓冲区读取数据的特性来对 exit 函数和_exit 函数进行比较的应用实例。

【例 7.13】展示 exit 和_exit 函数的区别

应用代码在子进程中调用 printf 函数输出一串没有换行符的字符串，由于没有换行符，printf 函数不会输出，此时调用 exit 将文件缓冲区的内容写回文件，所以在输出终端（屏幕）上可以看到该字符串，需要注意的是该字符串并不会换行；在父进程中调用 printf()做同样的操作，然后调用_exit 函数，此时_exit 函数会直接扔掉缓冲区的数据，所以看不到父进程的字符串输出。

实例的应用代码如下：

```
1   /*体现 exit 和_exit 的区别*/
2   #include <sys/types.h>
3   #include <unistd.h>
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <errno.h>
7   int main(void)
8   {
9       pid_t pid;
10      if ( (pid = fork() )==-1 )                  //如果创建子进程失败
11       {
12          perror ("创建子进程失败\n");            //创建子进程出错信息
13          exit(0);
14       }
15      else if(pid==0)                             //子进程
16       {
17          printf("01:这是子进程\n");
18          printf("02:这是子进程，目前数据在缓冲区中");
19          //这个地方没有换行符，所以不写出数据
20          exit(0);                                //退出，强制清空，会输出上面未完成的数据
21       }
22      else                                        //父进程
23       {
24          sleep(1);                               //休眠一秒以确定先后顺序
25          printf("03:这是父进程，开始输出\n");
```

```
26              printf("04:这是父进程，目前数据在缓冲区中");    //同样没有换行符
27              _exit(0);                                      //_exit 函数会直接丢弃相应的数据
28          }
29          return 0;
```

将文件保存为 exam713exit.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam713exit。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam713exit.c -o exam713exit
```

执行 exam713exit，可以看到如下的输出，代码中标号为 04 的字符串并没有输出，而是直接被 _exit 函数丢弃掉了。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam713exit
01:这是子进程
02:这是子进程，目前数据在缓冲区中
03:这是父进程，开始输出
```

在一个进程退出的时候，可能存在如下两种状态：

- 其父进程恰好忙于处理其他事务，不能接收子进程的终止状态，如果此时子进程完全消失了，那么当父进程处理完其他事务后想要检查子进程的情况时，就没有可用的信息了，所以 Linux 内核为每个已结束的进程保留一定的信息，一般至少包含进程标识符、终止状态字、进程处理器时间等信息；在任何时候父进程可以通过调用 wait 或者 waitpid 函数都能得到相应的数据，在此之后，Linux 内核再将保存这些信息的数据结构释放。通常把这种已经结束、但其父进程尚未检查其终止状态的进程称为僵尸进程。
- 如果父进程可能先于子进程结束，此时 init 进程就会自动成为该子进程的父进程。通常的实现机制是：当一个进程结束时，系统逐一检查所有的活动进程，如果某进程的父进程是这个被结束的进程，系统就将这个活动进程的父进程标识符置为 1，即 init 的进程标识符，这样就保证了每个进程都有它的父进程。


注　意

由以上可以知道当调用 exit 系列函数或者 return 函数返回的时候，其实进程并没有真正的完全消失，其还在继续占用部分资源；如果这种僵尸进程过多，则会大大影响系统的性能，在下一小节中将介绍如何处理僵尸进程。

exit 函数在进程退出的时候会自动调用一些函数对当前退出的进程进行相应的处理，这些函数通常被称为终止处理函数（exit handler），每个进程对应的终止处理函数最多可以达到 32 个（基于标准 C），如果希望将一个指定函数添加到这个终止处理函数中，可以使用 atexit 函数，对其标准调用格式说明如下：

```
#include <stdlib.h>
int atexit(void (*function)(void));
```

其参数 function 是一个函数的地址，当调用该指定函数的时候不需要向这个指定函数传送参数，也不期望有返回值；若调用 atexit 函数成功，则返回 0，如果调用 atexit 函数失败则会返回一

个非 0 值。

在使用 atexit 登记进程的终止处理函数时需要注意以下两点：

- 当进程退出的时候，如果存在多个使用 atexit 函数添加的终止处理函数，exit 函数调用这些终止处理函数的次序和使用 atexit 函数添加它们的顺序刚好相反。
- 如果一个函数被使用 atexit 函数登记了多次，则在进程退出的时候也会被调用多次。

例 7.14 是一个使用 atexit 登记两个用户自定义函数的实例。

【例 7.14】使用 atexit 登记终止处理函数

应用代码定义了两个终止处理函数 exitfun1 和 exitfun2，这两个函数分别用于调用 printf 函数在屏幕上输出一组字符串，然后在主函数 main 中调用了 atexit 函数对这两个函数进行了登记，其中 exitfun2 登记了两次，然后在主函数中输出了另外一个字符串后退出。

实例的应用代码如下：

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  //用于登记退出执行的第一个函数
 4  void exitfun1(void)
 5  {
 6    printf("这是第一个终止处理函数!\n");
 7    return;
 8  }
 9  //用于登记执行的第二个函数
10  void exitfun2(void)
11  {
12    printf("这是第二个终止处理函数!\n");
13    return;
14  }
15  int main(int argc,char *argv[])
16  {
17    atexit(exitfun1);                    //登记两个函数
18    atexit(exitfun2);
19    atexit(exitfun2);                    //再次登记
20    printf("这是主程序的输出!\n"); //在主程序中输出一个字符串
21    exit(0);
22  }
```

将文件保存为 exam714atexit.c，在 gcc 中进行编译链接，生成可执行文件 exam714atexit。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam714atexit.c -o exam714atexit
```

运行该可执行文件，可以看到首先输出的是 exitfun2 内的字符串，而且由于登记了两次，则输出了两次，然后才是 exitfun1 内的字符串。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam714atexit
这是主程序的输出!
```

```
这是第二个终止处理函数!
这是第二个终止处理函数!
这是第一个终止处理函数!
```

7.3.5　销毁进程

在上一节中介绍过当一个进程使用 exit 系列函数退出的时候，其会在内存中保留部分数据以供父进程查询，同时其也会产生一个终止状态字，然后 Linux 内核会发出一个 SIGCHLD 信号以通知父进程，因为子进程的结束对于父进程是异步的，因而这个 SIGCHLD 信号对于父进程也是异步的，父进程可以不响应。

父进程对于退出之后的子进程的默认状态是不处理的，事实上在以前给出的实例中也都没有处理，但是这样会导致系统中的僵尸进程浪费了系统资源，此时应该调用 wait 函数或 waitpid 函数对这些僵尸进程进行处理。

在调用 wait 或者 waitpid 函数之后可能存在如下三种情况：

- 如果该父进程的所有子进程都还在运行，则阻塞父进程自身以等待子进程的运行结束。
- 如果有一个子进程已经结束，则父进程取得该子进程的终止状态，并且立即返回。
- 如果该父进程没有任何子进程，则立即出错返回。

对 wait 和 waitpid 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

1. wait 函数

如果 wait 函数调用成功则返回子进程的标识符，如果失败则返回-1，其中参数 status 是一个整型指针，可以用于存放子进程的终止状态，也可以定义为一个空指针。

wait 函数和 waitpid 函数不同，在有一个子进程终止之前，wait 函数让父进程阻塞以等待子进程退出，而 waitpid 有一个参数可以让父进程不阻塞（将在下一小节中介绍），并且在一个父进程有多个子进程的情况下，如果其中有一个子进程退出则会返回该子进程的进程标识符，表 7.3 是 wait 函数返回的终止状态的宏。

表 7.3　wait 函数返回的宏

宏	说明
WIFEXITED(status)	当子进程正常结束时返回为真
WIFSIGNALED(status)	当子进程异常结束时返回为真
WEXITSTATUS(status)	当 WIFEXITED(status)为真时调用，返回状态字的低 8 位
WTERMSIG(status)	当 WIFSIGNALED(status)为真时调用，返回引起终止的信号代号

2. waitpid 函数

在使用 wait 函数的时候，如果父进程的任何一个子进程返回则 wait 函数返回，而 waitpid 函

数可以通过参数来指定需要等待的子进程。

waitpid 函数的参数 pid 用于对子进程进行相应的筛选，对其详细说明如下。

- pid > 0：只等待进程 ID 为 pid 的子进程，不管其他已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid 就一直等待下去。
- pid = −1：等待任何一个子进程退出，没有任何限制，此时 waitpid 等价于 wait。
- pid = 0：等待同一个进程组中的任何子进程，如果某一子进程已经加入了其他进程组，则 waitpid 不会对它做任何理睬。
- pid < −1：等待一个指定进程组中的任何子进程，这个进程组的 ID 等于 pid 的绝对值。

waitpid 函数的参数 options 用于进一步控制 waitpid 函数的操作，其可以是 0，也可以是 WNOHANG 和 WUNTRACED 两个选项之一，或者是使用“|”符号连接的“或”操作，对这两个关键字定义如下。

- WNOHANG：如果由 pid 指定的子进程并不是立即可用的，则 waitpid 函数不阻塞，此时返回“0”。
- WUNTRACED：如果某实现支持作业控制，而由 pid 指定的任意子进程已经处于暂停状态，并且未报告过，则返回其状态。

对于 waitpid 函数而言，如果指定的进程或者进程组不存在，或者参数 pid 指定的进程不是父进程所调用的子进程，都将出错。

总体而言，waitpid 函数提供了 wait 函数所没有的如下三个功能：

- 能够等待指定的一个进程结束。
- 能够不阻塞父进程获得子进程的状态。
- 支持作业控制（读者可以自行查阅相关的资料）。

例 7.15 是 waitpid 函数的一个应用实例。

【例 7.15】使用 waitpid 函数退出进程

应用代码在主程序中嵌套创建了子进程和孙进程，然后子进程退出成为僵尸进程，而孙进程休眠 2 秒之后再退出。

实例的应用代码如下：

```
1    #include <sys/types.h>
2    #include <sys/wait.h>
3    #include <stdio.h>
4    #include <errno.h>
5    #include <stdlib.h>
6    int main(void)
7    {
8            pid_t pid;
9            if((pid=fork())<0)                    //创建子进程失败
10           {
```

```
11                  perror("创建子进程失败.\n");       //创建子进程失败
12                  exit(0);
13              }
14              else if(pid==0)     //进入子进程
15              {
16                  if((pid=fork())<0)                  //在子进程中继续创建一个子进程
17                  {
18                          perror("创建子进程失败.\n");
19                          exit(0);
20                  }
21                  else if(pid>0)                      //当前创建子进程的父进程，即第一个子进程
22                          {
23                      exit(0);                        //退出第一个子进程
24                          }
25                          else
26                          {
27                    sleep(2);                         //休眠 2 秒
28                    printf("这是第二个子进程, parent pid=%d \n", getppid());
29                    exit(0);
30                          }
31              }
32
33              if(waitpid(pid, NULL, 0)!=pid)          //判断到底是哪个进程退出了
34              {
35                  perror("waitpid 销毁进程失败.\n");
36                  exit(0);
37              }
38              exit(0);
39          }
```

将文件保存为exam715waitpid.c，在终端中使用gcc编译链接，生成可执行文件exam715waitpid。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam715waitpid.c -o exam715waitpid
```

运行该可执行文件，可以看到在退出了第 2 个子进程后会停止运行一直等待，此时可以使用 Ctrl+C 组合键退出。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam715waitpid
alloy@ubuntu:~/linuxc/chapter7$  这是第 2 个子进程, parent pid=1
```


注　意

综上所述，wait 系列函数的作用主要是完全销毁进程以释放内存以及获得进程的退出状态；除了 wait 函数和 waitpid 函数之外，Linux 内核还提供了 wait3、wait4 和 waitid 等函数，读者可以自行参考相应的手册。

7.3.6　Linux 的进程操作总结

对 Linux 的进程操作过程可以总结如下：

01 调用 fork 函数创建一个新的进程。

02 分别在子进程和父进程中进行相应的操作，可以继续调用 fork 函数创建一个新的进程，也可以调用 exec 函数族进行相应的操作，然后调用 exit 函数族退出相应的进程。

03 调用 wait 函数族销毁进程。

例 7.16 是一个典型的完整进程操作的应用实例。

【例 7.16】一个完整进程操作

父进程首先通过 fgets 函数从用户输入终端（键盘）读取一个要执行的命令，然后创建一个子进程来执行这个命令，并且将该命令的执行返回值返回给父进程，在父进程中打印输出该返回值，其流程如图 7.13 所示。

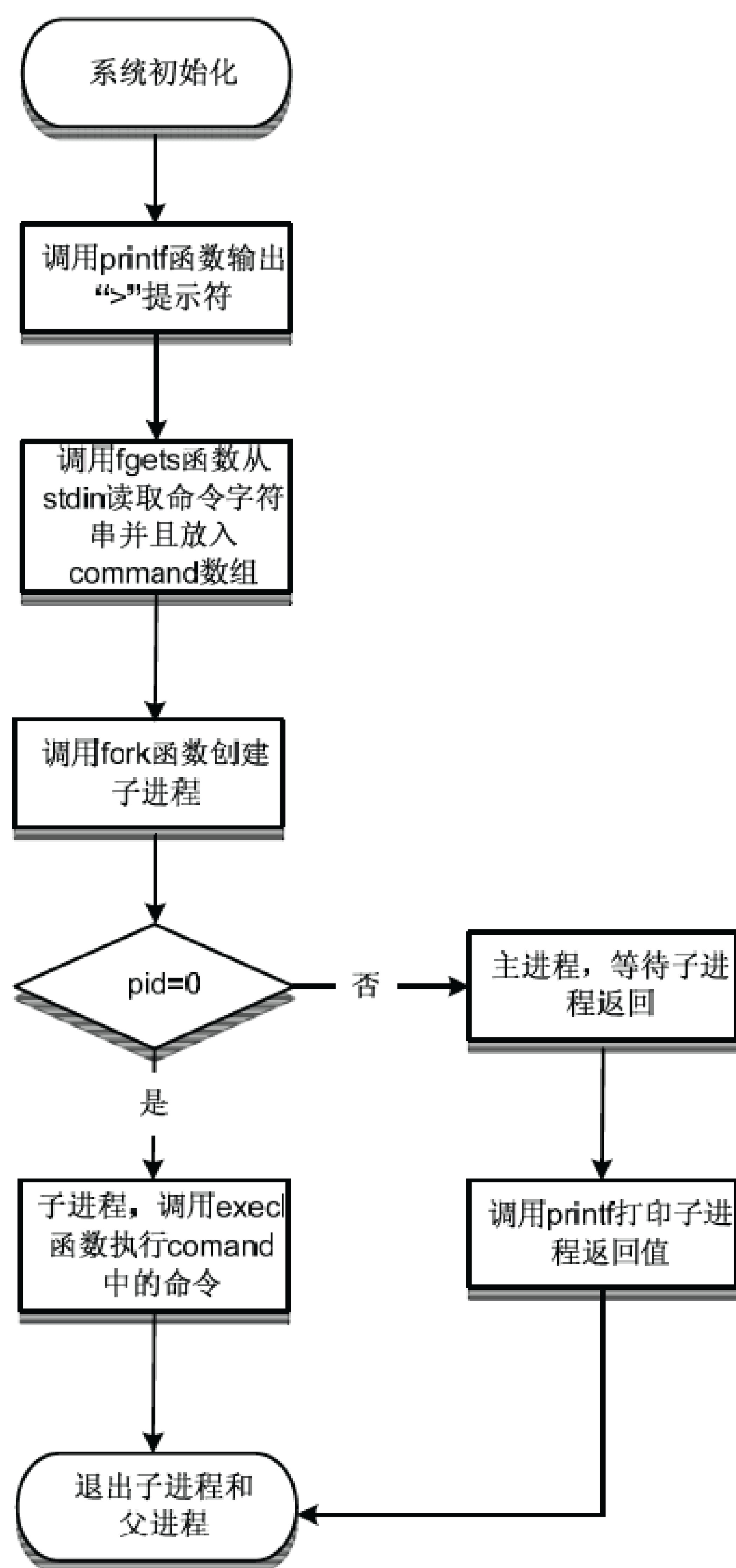


图 7.13 进程操作实例流程

实例的应用代码如下：

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <sys/types.h>
4    #include <unistd.h>
5    #include <sys/wait.h>
```

```
 6  #include <string.h>
 7  #include <error.h>
 8  char command[256];
 9  void main()
10  {
11      int rtn;                        /*子进程的返回数值*/
12      while(1)
13      {
14          printf( ">" );              //从终端读取要执行的命令
15          fgets(command, 256, stdin); //将命令数据存放到 command 中
16          command[strlen(command)-1] = 0;
17          if (fork() == 0 )           //在子进程中执行这个命令
18          {
19              execlp(command,command,NULL);
20              //如果 exec 函数返回，表明没有正常执行命令，打印错误信息
21              perror(command);
22              exit(1);
23          }
24          else    //在父进程中等待子进程结束，并且打印子进程的返回值
25          {
26              wait( &rtn );
27              printf("子进程返回%d\n",rtn);
28              exit(0);
29          }
30      }
31      exit(0);
32  }
```

将文件保存为 exam716fullprocess.c，在终端中使用 gcc 编译链接，生成可执行文件 exam716fullprocess。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam716fullprocess.c -o exam716fullprocess
```

在当前工作目录中运行该可执行文件，然后在其中调用 date 命令，可以看到打印出对应的日期信息，然后从子进程中退出。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam716fullprocess
>date
2014 年 02 月 27 日 星期四 09:55:00 CST
子进程返回 0
```

7.4　进程综合应用——使用多个进程创建文件

例 7.17 是一个使用两个进程分别调用第 3 章中的实例 3.14 和 3.16 对应的可执行文件来创建一个以当前时间为文件名的文件，以及将一个与当前时间相关的字符串连续写入一个文件的实例，对这两个可执行文件的说明如下。

- exam314ConWriteTimeFun: 这是一个将当前时间对应的字符串连续写入一个指定文件的

可执行文件，需要提供一个文件名作为指定文件的文件名。

- exam316timeOpen：这是一个使用当前时间对应的字符串作为文件名来新建一个文件的可执行文件。

应用代码在父进程中创建了子进程 1 和子进程 2，并且在子进程 1 中调用了 exam314ConWriteTimeFun，在子进程 2 中调用了 exam316timeOpen，其流程如图 7.14 所示。

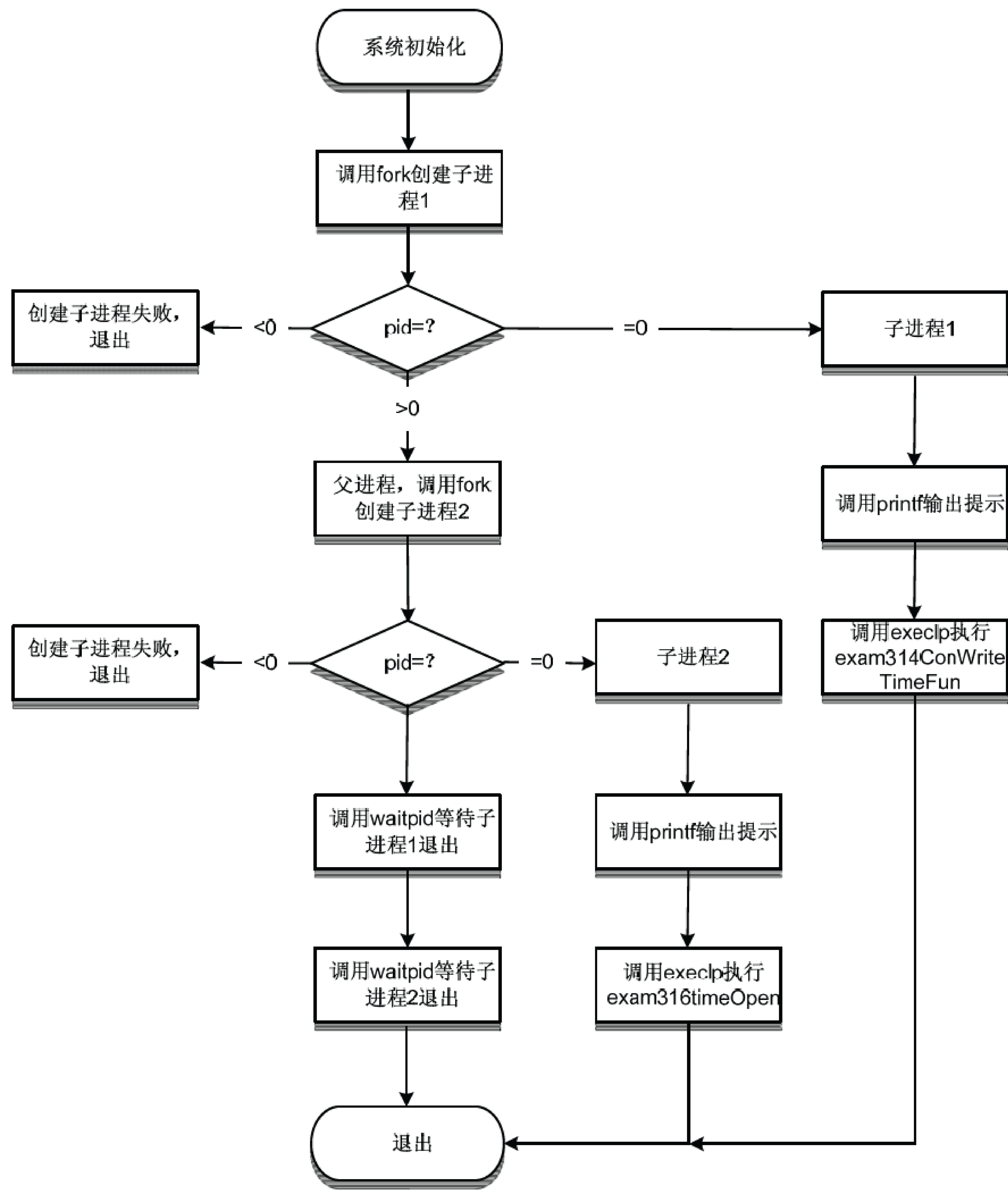


图 7.14　进程综合应用实例流程

实例的应用代码如下：

【例 7.17】使用多个进程创建文件

```
1	#include <stdio.h>
2	#include <stdlib.h>
3	#include <sys/types.h>
4	#include <unistd.h>
```

```
5     #include <sys/wait.h>
6     int main(void)
7     {
8        pid_t child1,child2,child;                   //进程标识符
9        child1 = fork();                             //创建第一个子进程
10       if (child1 < 0)                              //创建第一个子进程失败
11       {
12           printf("创建第一个子进程失败!\n");
13           exit(1);
14       }
15       else if(child1 == 0)   //子进程 1
16       {
17           printf("这是第一个子进程:\n");
18           if(execlp("./exam314ConWriteTimeFun","./exam314ConWriteTimeFun",
             "timerecondtxt",NULL)<0)                 //调用创建文件的程序
19           {
20              printf("调用创建文件程序失败！ \n");
21           }
22       }
23       else                                         //以下为父进程
24       {
25          child2 = fork();                          //创建子进程 2
26          if(child2 < 0)                            //如果创建子进程 2 失败
27          {
28             printf("创建第二个子进程失败!\n");
29             exit(1);
30          }
31          else if(child2 == 0)
32          {
33             printf("这是第二个子进程！ \n");
34             if(execlp("./exam316timeOpen","./exam316timeOpen",NULL)<0)   //调用创建文件的程序
35             {
36                printf("调用创建文件程序失败！ \n");
37             }
38          }
39          printf("这是父进程\n");
40          child = waitpid(child1,NULL,0);                //等待进程 1 退出
41          if(child == child1)                            //进程 1 退出
42          {
43              printf("进程 1 已经退出!\n");
44          }
45          child = waitpid(child2,NULL,0);                //等待进程 2 退出
46          if(child == child2)                            //进程 2 退出
47          {
48              printf("进程 2 已经退出\n");
49          }
50       }
51       exit(0);                                          //退出
52    }
```

将文件保存为 exam717processfile.c，使用 gcc 在终端中编译链接，生成可执行文件 exam717processfile。

```
alloy@ubuntu:~/Cexam/chapter8$ gcc exam717processfile.c -o exam717processfile
```

将 exam314ConWriteTimeFun 参数中的文件名设置为“timerecondtxt”，运行后可以看到如下的输出。

```
alloy@ubuntu:~/Cexam/chapter8$ ./exam717processfile        //运行
这是父进程
这是第一个子进程:
这是第二个子进程!                                            //两个进程的提示
当前时间为 Wed Jul 31 03:47:44 2013
//在进程 2 中调用，创建一个名为 File114744 的文件
11:47:44
step1 timebuf is 114744
writetimebuf is 114744
step2 timebuf is
filenamebuf is File114744
//以下是子进程 1 中的调用，以下字符串被写入 timerecondtxt 文件
Wed Jul 31 11:47:44 2013
Wed Jul 31 11:47:45 2013
Wed Jul 31 11:47:46 2013
Wed Jul 31 11:47:47 2013
^C    //使用 Ctrl+C 键退出
```

使用 cat 命令打开 timerecondtxt 文件，可以看到其中写入的字符串。

在例 7.17 中存在如下两个问题：

- 子进程 1 和子进程 2 的执行先后顺序并不确定。
- 如果子进程 1 和子进程 2 同时对同一个文件进行操作，例如使用子进程 2 负责创建文件，而子进程 1 负责将一个字符串写入子进程 2 刚刚创建的文件中，则可能出现错误。

以上的两个问题可以通过进程的同步机制来解决，本书将在后续章节中对解决方法进行详细介绍。

7.5 Linux 的进程组和会话

在 Linux 中，可以把若干个进程组合成一个集合，而这些集合又可以再组合成集合，这就引出进程组和会话的概念，进程组和会话都有一些特性，其中控制终端是最重要的特性之一。

1. 进程组

进程组是若干个进程的集合，每个进程除了拥有一个进程 ID 之外，还隶属于一个进程组。进程组是一个或多个进程的集合。每个进程组都有一个唯一的进程组 ID。进程组 ID 类似于进程 ID——它是一个正整数，并可存放在 pid_t 的数据类型中。函数 getpgrp 返回调用进程的进程组 ID：

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp (void);
```

函数没有参数，当调用其时，返回调用进程的进程组 ID。

每个进程组都有一个组长进程 leader。组长进程 leader 的标识是：其进程组 ID 等于其进程 ID。该进程组组长可以创建一个进程组，也可以创建该组中的进程，然后终止。只要在某个进程组中有一个进程存在，则该进程组就存在，这与其组长进程是否终止无关。从进程组创建开始到其中最后一个进程离开为止的时间区间称为进程组的生命期。某个进程组中的最后一个进程可以终止，也可以参加另一个进程组。

进程调用 setpgid 可以加入一个现存的组或者创建一个新进程组（下一节中将说明利用 setsid 也可以创建一个新的进程组）。

```
#include <sys/types.h>
#include <unistd.h>
int setpgid (pid_t pid, pid_t pgid);
```

若调用成功则返回 0，若出错则返回-1。

函数将 pid 进程的进程组 ID 设置为 pgid。如果这两个参数相等，则由 pid 指定的进程变成进程组组长。

一个进程只能为自己或它的子进程设置进程组 ID，当在其子进程中调用了 exec 函数之后，它就不能再改变该子进程的进程组 ID 了。

如果 pid 是 0，则使用调用者的进程 ID。另外，如果 pgid 是 0，则由 pid 指定的进程 ID 作为进程组 ID。

在大多数作业控制 shell 中，在 fork 之后调用此函数，使父进程设置其子进程的进程组 ID，然后使子进程设置其自己的进程组 ID。这些调用中有一个是冗余的，但这样做可以保证父、子进程在进一步操作之前，子进程都进入了该进程组。如果不这样做的话，那么就产生一个竞态条件，因为它依赖于哪一个进程先执行。

在讨论信号时，将说明如何将一个信号发送给一个进程（由其进程 ID 标识）或发送给一个进程组（由进程组 ID 标识）。同样，waitpid 则可被用来等待一个进程或者指定进程组中的一个进程。

2. 会话

会话是若干个进程组的集合，会话可包含多个进程组，但只能有一个前台进程组。

图 7.15 是一个会话的示意图，其由三个进程组组成，其中进程 proc1 和 proc2 属于同一个后台进程组，进程 proc3、proc4、proc5 属于同一个前台进程组，shell 进程本身属于一个单独的进程组。

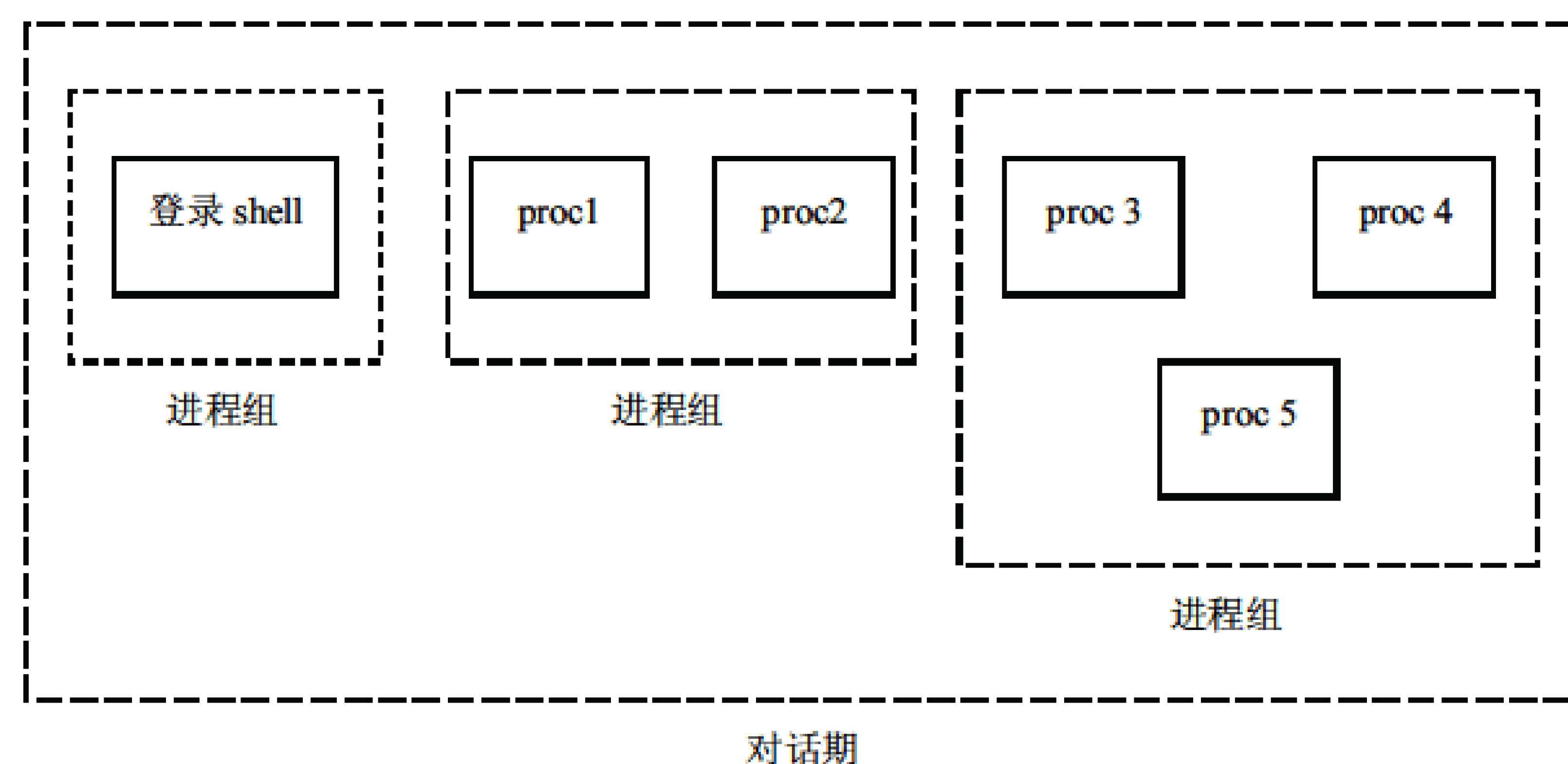


图 7.15　一个会话的示意图

这些进程组的控制终端相同，它们属于同一个进程组。当用户在控制终端输入特殊的控制键（例如 Ctrl+C）时，内核会发送相应的信号（例如 SIGINT）给前台进程组的所有进程。

这种安排通常是由 shell 的管道线将几个进程编成一组的。例如，图 7.14 中的安排可能是由下列形式的 shell 命令形成的：

```
procl | proc2 &
proc3 | proc4 | proc5
```

进程调用 setsid 函数就可建立一个新对话期，setsid 的函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t setsid (void);
```

若调用成功则返回进程组 ID，若调用出错则返回-1。

如果调用此函数的进程不是一个进程组的组长进程 leader，则此函数创建一个新的会话期，对其结果说明如下。

- 此进程变成该新会话期的会话期首进程（session leader，会话期首进程是创建该会话期的进程），此进程是该新会话期中的唯一进程。
- 此进程成为一个新进程组的组长进程，新进程组 ID 是此调用进程的进程 ID。
- 此进程没有控制终端（下一节讨论控制终端），如果在调用 setsid 之前此进程有一个控制终端，那么这种联系也被解除。

如果此调用进程已经是一个进程组的组长，则此函数返回出错。为了保证不处于这种情况，通常先调用 fork，然后使其父进程终止，而子进程则继续。因为子进程继承了父进程的进程组 ID，而其进程 ID 是新分配的，两者不可能相等，所以这就保证了子进程不是一个进程组的组长。

例 7.18 给出了一个从会话和进程组的角度来分析 Linux 系统登录和执行命令的过程实例。

【例 7.18】Linux 系统的登录过程

01 getty 或 telnetd 进程在打开终端设备之前调用 setsid 函数创建一个新的会话，该进程称为会话首进程（Session Leader），该进程的 ID 也可以看作 session 的 ID，然后该进程打开终端设备作为这个会话中所有进程的控制终端。在创建新会话的同时也创建了一个新的进程组，该进程是这个进程组的组长进程（Process Group Leader），该进程的 ID 也是进程组的 ID。

02 在登录过程中，getty 或 telnetd 进程变成 login，然后变成 shell，但仍然是同一个进程，仍然是 Session Leader。

03 由 shell 进程 fork 出的子进程本来具有和 shell 相同的会话期、进程组和控制终端，但是 shell 调用 setpgid 函数将作业中的某个子进程指定为一个新进程组的 Leader，然后调用 setpgid 将该作业中的其他子进程也转移到这个进程组中。如果这个进程组需要在前台运行，就调用 tcsetpgrp 函数将它设置为前台进程组，由于一个 session 只能有一个前台进程组，所以 shell 所在的进程组就自动变成后台进程组。

在如图 7.15 所示的例子中，proc3、proc4、proc5 被 shell 放到同一个前台进程组，其中有一个

进程是该进程组的 Leader，Shell 调用 wait 等待它们运行结束。一旦它们全部运行结束，shell 就调用 tcsetpgrp 函数将自己提到前台继续接受命令。但是注意，如果 proc3、proc4、proc5 中的某个进程又 fork 出子进程，子进程也属于同一进程组，但是 Shell 并不知道子进程的存在，也不会调用 wait 等待它结束。换句话说，proc3、proc4、 proc5 是 shell 的作业，而这个子进程不是，这是作业和进程组在概念上的区别。一旦作业运行结束，shell 就把自己提到前台，如果原来的前台进程组还存在（如果这个子进程还没终止），则它自动变成后台进程组。

3. 会话和进程组的特性

每个会话都有一个控制终端，这是会话最重要的特性之一，当用户登录 Linux 系统时，将自动建立控制终端（Controlling Terminal）。对控制终端进行读写的方法是打开文件/dev/tty（设备文件），在内核中，此特殊文件是控制终端的同义语。通常情况下，对于标准输入、标准输出以及标准出错，程序都要与控制终端实现交互。

会话和进程组有一些其他特性，如图 7.16 所示。

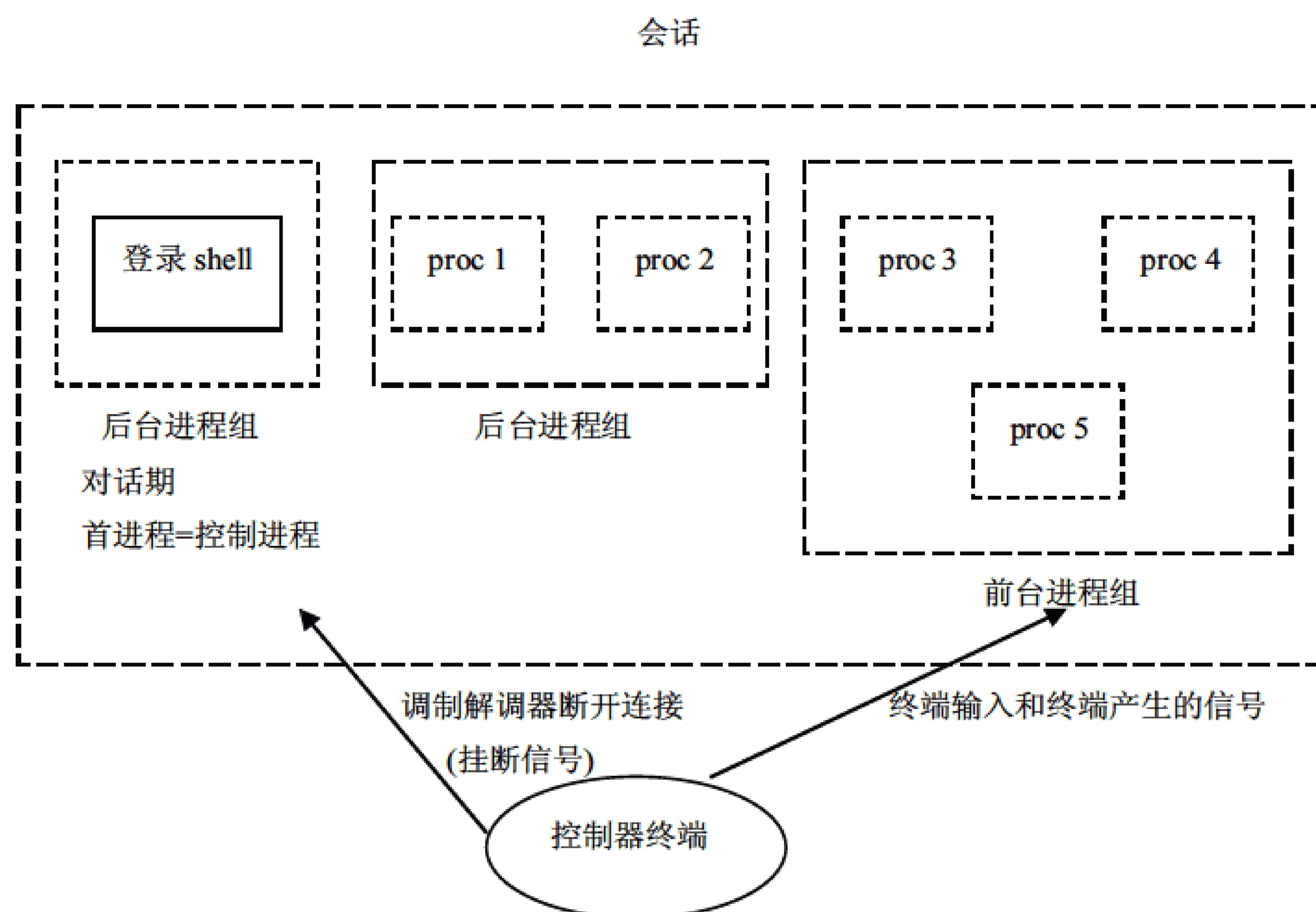


图 7.16　进程组和会话的特性

- 一个会话可以有一个单独的控制终端。这通常是我们在其上登录的终端设备（终端登录情况）或伪终端设备（网络登录情况）。
- 建立与控制终端连接的会话首进程，称之为控制进程（Controlling Process）。
- 一个会话期中的几个进程组可被分成一个前台进程组（Foreground Process Group）以及一个或几个后台进程组（Background Process Group）。
- 如果一个会话期有一个控制终端，则它有一个前台进程组，其他进程组则为后台进程组。
- 无论何时键入中断键（通常是 Delete 或 Ctrl+C）或退出键（通常是 Ctrl+\），就会造成将中断信号或退出信号送至前台进程组的所有进程。
- 如果终端界面检测到调制解调器已经脱开连接，则将挂断信号送至控制进程（会话期首进程）。

在一个会话中仅有一个前台进程组，那么就需要有一种方法来通知内核哪一个进程组是前台进程组，这样，终端设备驱动程序就能了解将终端输入和终端产生的信号送到何处，如图 7.15 所示。

相关的系统调用为 tcgetpgrp 和 tcsetpgrp。tcgetpgrp 返回前台进程组的 ID，函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t tcgetpgrp (int fd);
```

如果函数调用成功则返回前台进程组 ID，若出错则返回“-1”。

tcsetpgrp 用于将某一进程组设置为前台进程组，函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
int tcsetpgrp (int fd, pid_t pgrpid);
```

如果函数调用成功则返回“0”，若出错则返回“-1”。

函数 tcgetpgrp 返回前台进程组 ID，它与在文件上打开的终端相关。

如果进程有一个控制终端，则该进程可以调用 tcsetpgrp 将前台进程组 ID 设置为 pgrpid。pgrpid 值应当是在同一会话期中的一个进程组 ID。fd 必须引用该会话期的控制终端。大多数应用程序并不直接调用这两个函数，它们通常由作业控制 Shell 调用。

7.6 Linux 进程的其他操作

Linux 进程的其他操作包括更改进程的用户、在进程中调用其他应用程序、统计进程状态和进程的执行时间。

7.6.1 更改进程用户

在第 7.2.2 小节中介绍过，进程和文件类似，也有对应的实际用户 ID 等属性，与某一个进程相关联的 ID 至少有 6 个，它们是：实际用户 ID、实际组 ID、有效用户 ID、有效组 ID、保存的设置用户 ID 和保存的设置组 ID，并且，通常情况下，有效用户 ID 等于实际用户 ID，有效组 ID 等于实际组 ID。这些 ID 都和进程的安全性息息相关，其中最关键、最常用的是进程的用户 ID 和其用户组 ID。

在 Linux 操作系统中，从安全性出发，通常是给一个进程最小的权限（最小特权 least privilege 模型），但是这种权限有可能在访问系统资源或者进行一些操作的时候因为没有足够的权限失败，此时可以通过更换进程的用户 ID 或组 ID，使得新的 ID 具有适合的特权或访问权限。相反，当需要降低其特权或阻止对某些资源的访问时，也需要更换用户 ID 或组 ID，从而使得新 ID 不具有相应特权或访问这些资源的能力。Linux 内核提供了一系列的函数来完成对应的功能。

1. setuid 函数和 setgid 函数

setuid 函数和 setgid 函数分别用于设置实际用户 ID、有效用户 ID、实际组 ID、有效组 ID，对

其标准调用格式说明如下：

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

函数的参数即为期望设置的目标 ID，如果函数调用成功则返回“0”，若出错则返回“-1”。

需要注意的是，并不是每个用户 ID 都有权限修改一个进程的这些参数，其必须遵循如下的规则（同时针对用户 ID 和组 ID）：

- 若进程具有超级用户特权，则 setuid 函数将实际用户 ID、有效用户 ID，以及保存的设置用户 ID 设置为 uid（对于 setgid 则为 gid）。
- 若进程没有超级用户特权，但是 uid 等于实际用户 ID 或保存的设置用户 ID，则 setuid 只将有效用户 ID 设置为 uid（对于 setgid 则为 gid），不改变实际用户 ID 和保存的设置用户 ID。
- 如果上面两个条件都不满足，则 errno 设置为 EPERM，并返回出错。

此外，关于 Linux 内核所维护的三个用户 ID，还需要注意如下几点：

- 只有超级用户进程可以更改实际用户 ID。通常，实际用户 ID 是在用户登录时，由 login(1) 程序设置的，而且决不会改变它。因为 login 是一个超级用户进程，当它调用 setuid 时，设置所有三个用户 ID。
- 仅当对程序文件设置了用户 ID 位时，exec 函数设置有效用户 ID。如果用户 ID 位没有设置，则 exec 函数不会改变有效用户 ID，而将其维持为原先的值。任何时候都可以调用 setuid，将有效用户 ID 设置为实际用户 ID 或保存的设置用户 ID。自然，不能将有效用户 ID 设置为任一随机值。
- 保存的设置用户 ID 是由 exec 从有效用户 ID 复制的。在 exec 按文件用户 ID 设置了有效用户 ID 后，即进行这种复制，并将此副本保存起来。

表 7.4 给出了修改这三个用户 ID 的不同方法。

表 7.4 修改三个用户 ID 的不同方法

ID	exec 系列函数		setuid 函数	
	设置用户 ID 位关闭	设置用户 ID 位打开	超级用户	非特权用户
实际用户 ID	不变	不变	设为 uid	不变
有效的用户 ID	不变	设置为程序文件的用户 ID	设为 uid	设为 uid
保存的设置用户 ID	从有效用户 ID 复制	从有效用户 ID 复制	设为 uid	不变

2. setregid 函数和 setregid 函数

setregid 函数和 setregid 函数用于交换实际用户 ID 和有效用户 ID 的值，对其标准调用格式说

明如下：

```
#include <sys/types.h>
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

如果函数调用成功则返回“0”，若出错则返回“-1”。

参数 ruid 和 rgid 表示实际用户 ID 和实际组 ID，euid 和 egid 表示有效用户 ID 和有效组 ID。setreuid 和 setregid 的作用很简单：一个非特权用户总能交换实际用户（或组）ID 和有效用户（或组）ID。这就允许一个设置用户 ID 程序转换成只具有用户的普通许可权，以后又可再次转换回设置用户 ID 所得到的额外许可权。

3. seteuid 函数和 setegid 函数

seteuid 函数和 setegid 函数用于更改有效用户 ID 和有效组 ID，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <unistd.h>
int seteuid (uid_t euid);
int setegid (gid_t egid);
```

如果函数调用成功则返回“0”，若出错则返回“-1”，其参数是希望设置的值。

图 7.17 是对以上介绍的各个函数的一个总结。

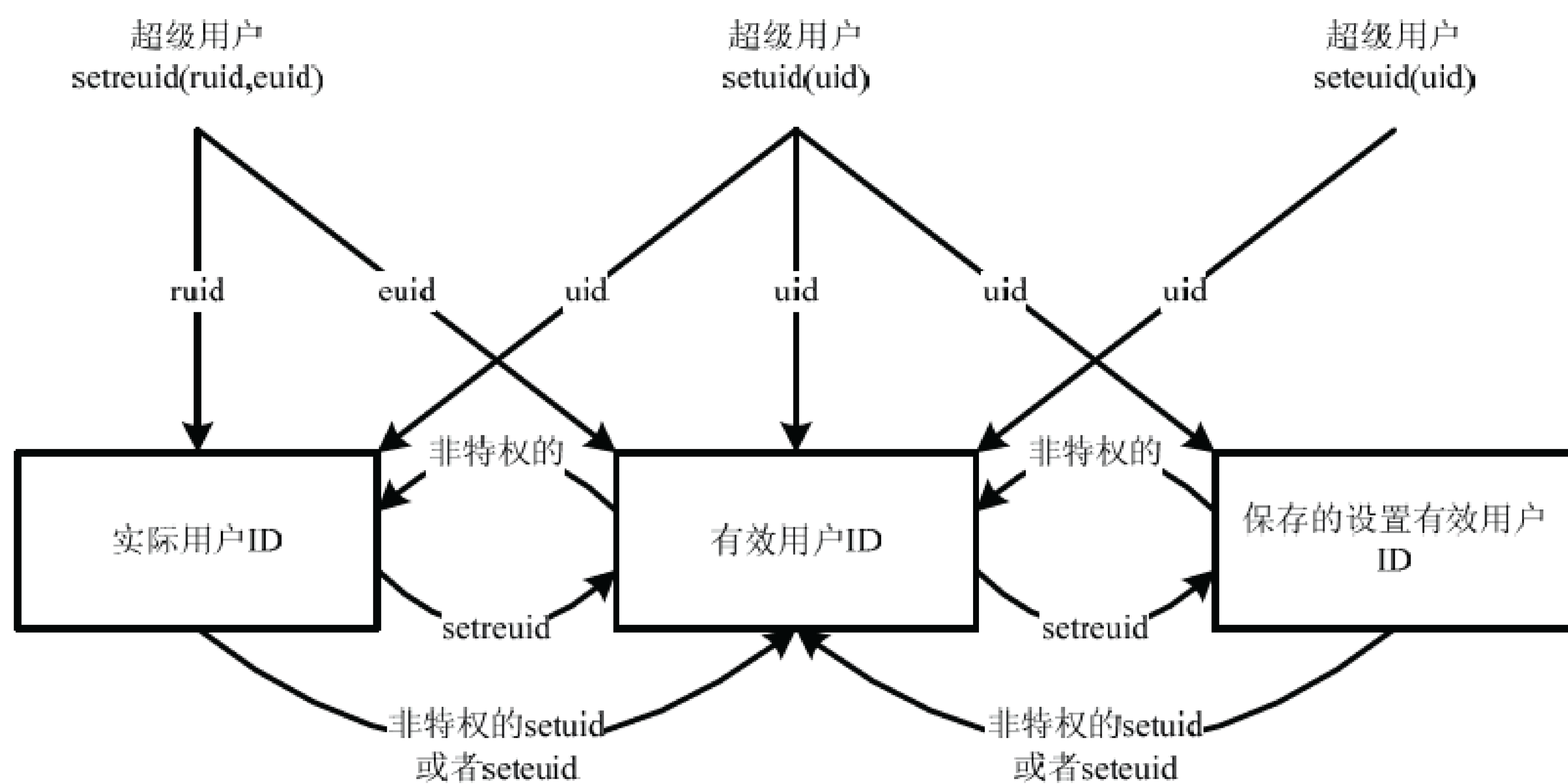


图 7.17　设置不同用户 ID 的函数



注　意

本小节中所提到的一切操作都适用于各个组 ID。

7.6.2　在进程中调用其他应用程序

例 7.7~例 7.9 展示了在进程中使用 exec 函数族来调用 ls 等命令的方法，但是这种方法首先要

使用 fork 函数来建立一个进程，在调用完成之后还需要调用 exit 和 wait 等相关函数来对进程进行退出和销毁；对于 Linux 内核提供的大量现成的命令，在进程中也可以使用 system 函数对其进行调用，此时不再使用这些相关的进程操作函数。

对 system 函数的标准调用格式说明如下：

```
#include <stdlib.h>
int system(const char *command);
```

system 函数的参数是需要调用的命令字符串，由于其内部调用了 fork 函数、exec 函数和 waitpid 函数，所以可能存在如下三种返回值：

- 如果 fork 失败或者 waitpid 返回除 EINTR 之外的出错，则 system 返回“-1”，而且 errno 中设置了错误类型。
- 如果 exec 函数调用失败（表示不能执行 shell），则其返回值如同 shell 执行了 exit（127）一样，即返回值为“127”。
- 如果所有三个函数（fork、exec 和 waitpid 函数调用）都成功，并且 system 的返回值是 shell 的终止状态。

对 system 函数的执行步骤说明如下：

01 调用 fork 函数产生子进程。
02 子进程调用/bin/sh-c cmdstring 来执行参数 cmdstring 字符串所代表的命令。
03 执行完成 cmdstring 后随即返回原调用的进程。


注　意

在调用 system 期间 SIGCHLD 信号（将在对应的信号章节中进行介绍）会被暂时搁置，SIGINT 和 SIGQUIT 信号则会被忽略。

例 7.19 是一个 system 函数的应用实例

【例 7.19】system 函数的应用

应用代码分别调用了 system 函数引用“date”和“who”命令来在屏幕上输出当前的时间信息以及登录用户的信息，这两个命令的参数直接以字符串的形式传递给 system 函数，此外还调用了一个名称为“nosuchcommand”的不存在命令用于展示 system 函数对于错误命令的处理，其操作流程如图 7.18 所示。

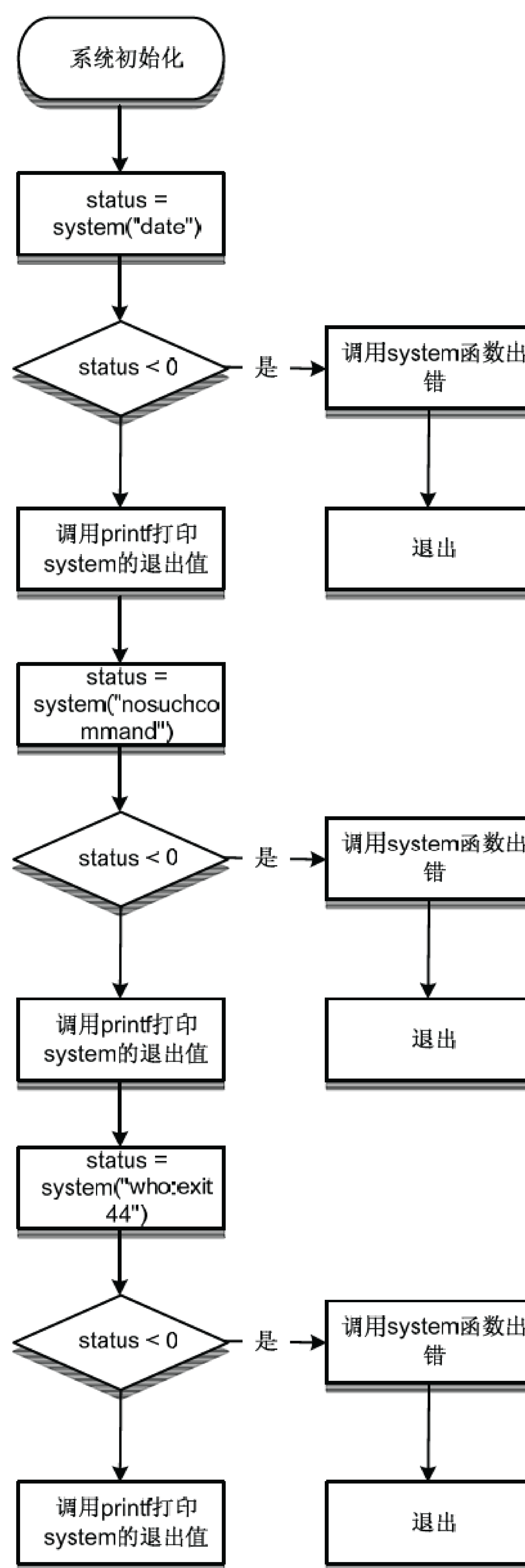


图 7.18　system 函数的应用实例

实例的应用代码如下：

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <errno.h>
4    int main(void)
5    {
6        int status;                                        //存放 system 函数的返回值
7        printf("system 函数调用 date 命令.\n");
```

```
 8      status = system("date");                         //调用 date 命令获得时间相关信息
 9      if(status <0)                                    //如果 status 小于 0，表明调用出错
10      {
11          perror("system 函数调用 date 出现错误.\n");     //调用出错
12          exit(0);
13      }
14      printf("system 函数的退出值是%d \n",status);        //输出 system 函数的返回值
15      printf("system 函数调用 nosuchcommand 命令.\n");
16      if((status=system("nosuchcommand"))<0) //如果 status 为 nosuchcommand 的返回值
17      {
18          printf("system 函数调用 nosuchcommand 错误");
19          exit(0);
20      }
21      printf("system 函数的退出值是%d \n",status);        //打印对应状态
22      printf("system 函数调用 who 命令\n");
23      if((status=system("who; exit 44"))<0) //调用 who 函数
24      {
25          perror("system 函数调用 who 出现错误");         //打印错误信息
26          exit(0);
27      }
28      printf("system 函数的退出值是%d\n",status);         //打印退出状态
29      exit(0);
30  }
```

将文件保存为 exam719system.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam719system。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam719system.c -o exam719system
```

执行该可执行文件，可以看到如下的输出，首先被调用的是 date 命令，在屏幕上输出当前的时间信息；然后 system 函数调用不存在的命令，shell 会提示该文件不存在；最后 system 函数调用了 who 命令，用于打印当前的登录用户信息。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam719system
system 函数调用 date 命令.
2014 年 02 月 27 日 星期四 10:37:42 CST
system 函数的退出值是 0
system 函数调用 nosuchcommand 命令.
sh: 1: nosuchcommand: not found
system 函数的退出值是 32512
system 函数调用 who 命令
alloy     tty7          2014-02-26 10:12
alloy     pts/1         2014-02-26 12:03 (alloy-mbp-2.local)
system 函数的退出值是 11264
```

7.6.3　统计进程状态

在 Linux 操作系统中，有些时候需要对当前的进程状态进行统计，例如获得所使用的 CPU 时间总量、用户 ID 和组 ID、启动时间等，这种功能被称为进程会计（Process Accounting），本小节

将介绍 Linux 下的这个命令以及对应的函数。

1. accton 命令

accton 函数可以用于对应的进程统计，但是在 Ubuntu 中其是不自带的，需要用户自己手动安装 acct 应用软件，如下所示：

```
alloy@ubuntu:/$ sudo apt-get install acct
[sudo] password for alloy:
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
下列【新】软件包将被安装：
  acct
……    //此处省略了部分自动安装的内容
正在处理用于 ureadahead 的触发器...
正在设置 acct (6.5.5-1ubuntu1) ...
Turning on process accounting, file set to '/var/log/account/pacct'.
 * Done.
```

使用 accton-V 命令，可以查看 accton 命令的版本号：

```
alloy@ubuntu:/$ accton -V
accton: GNU Accounting Utilities (release 6.5.5)
```

使用“accton on”命令即可打开进程的会计功能，其会在 var/log/account/的 pacct 文件中保存相应的进程信息，需要注意的是这个命令需要超级用户权限（可以使用 sudo 命令）：

```
alloy@ubuntu:/$ sudo accton on
Turning on process accounting, file set to the default '/var/log/account/pacct'.
```

在开始统计相应的信息之后，使用 lastcomm 命令可以看到如下的进程信息（只列举了部分内容）。

```
alloy@ubuntu:/$ lastcomm
sudo            S     alloy    pts/0    0.00 secs Tue Aug 26 08:29
accton          S     root     pts/0    0.00 secs Tue Aug 26 08:29
accton                root     pts/0    0.00 secs Tue Aug 26 08:29
accton                alloy    pts/0    0.00 secs Tue Aug 26 08:28
accton                alloy    pts/0    0.00 secs Tue Aug 26 08:28
……    //此处省略了部分列举内容
touch                 root     pts/0    0.00 secs Tue Aug 26 08:27
dpkg                  root     pts/2    0.03 secs Tue Aug 26 08:27
acct.postinst         root     pts/2    0.00 secs Tue Aug 26 08:27
invoke-rc.d           root     pts/2    0.00 secs Tue Aug 26 08:27
acct                  root     pts/2    0.00 secs Tue Aug 26 08:27
accton          S     root     pts/2    0.00 secs Tue Aug 26 08:27
```

使用 accton off 可以关闭进程会计，需要注意的是这个命令同样需要使用超级用户权限。

```
alloy@ubuntu:/$ sudo accton off
Turning off process accounting.
```

2. acct 函数

除了 accton 命令之外，Linux 内核还提供了一个函数 acct 用于实现相同的功能，对其标准调用格式说明如下：

```
#include <unistd.h>
int acct(const char *filename);
```

该函数的参数是一个用于记录进程信息的文件名称，如果调用成功则返回“0”，如果调用失败，则返回“-1”，同时设置相应的错误标志。


注 意

进程的相应信息是以一个结构体的格式记录在相应的文件中的，这个头文件会被定义在 sys/acct.h 文件中，读者可以自行参考相应的书籍。

7.6.4 进程的执行时间

一个进程在 Linux 中的执行是需要占用一定时间的，本小节将介绍如何测量一个进程的运行时间。

在 Linux 中有两种不同的时间计算方式：第一种是日历时间，这是自 1970 年 1 月 1 日 0 时 0 分 0 秒以来国际标准时间（UTC）所经过的秒数累积值，使用 time_t 数据类型用于表示这种时间值；第二种是进程时间（CPU 时间），是用于度量进程所使用的中央处理器的资源，其单位是时钟滴答，使用数据类型 clock_t 来表示这种时间值。

当需要测量一个进程的执行时间时，Linux 内核通常会使用如下三个时间值。

- 时钟时间：又称为墙上时钟时间（wall clock time），是进程运行的时间总量，其值和系统中同时运行的进程数有关。
- 用户 CPU 时间：执行用户指令所用的时间。
- 系统 CPU 时间：为该进程执行内核程序所经历的时间。

1. time 命令

可以使用 time 加上需要测量运行时间的命令来获得相应的执行时间，其使用说明如下，分别为使用 time 来测量 who 系统命令，和如例 7.2 所示的使用 fork 函数创建一个子进程对应的可执行文件，然后分别列出所用的时钟时间（real）、用户 CPU 时间（user）和系统 CPU 时间（sys）。

```
alloy@ubuntu:~/Cexam/chapter8$ time who
alloy       tty7            2013-08-01 19:33 (:0)
alloy       pts/1           2013-08-01 20:20 (192.167.112.1)

real   0m0.002s
user   0m0.000s
sys    0m0.004s
```

```
alloy@ubuntu:~/Cexam/chapter8$ time ./exam82fork
这是父进程，进程标识符是 4588
这是子进程，进程标识符是 4589

real   0m0.003s
user   0m0.000s
sys    0m0.000s
```

2. 如何取得时钟滴答数

在前面提到的进程时间是按照时钟滴答来计算的，但是时钟滴答和具体的系统相关，一秒钟可能是 50、60 或者 100 个滴答，可以使用 sysconf 函数来取得当前系统的时钟滴答数，其标准调用格式如下：

```
#include <unistd.h>
long sysconf(int name);
```

其中参数 name 有很多可选项，当需要获得时钟滴答数的时候应该使用_SC_CLK_TCK 宏定义作为参数，函数的返回值即为时钟的滴答数。

3. times 函数

在 Linux 中，用户可以调用 times 函数来计算当前进程自身的以及已经终止的进程执行时间，对其标准调用格式说明如下：

```
#include <sys/times.h>
clock_t times(struct tms *buf);
```

函数的参数指向一个 tms 结构，对该结构的定义说明如下，当函数调用成功的时候返回该进程执行所耗费的墙上时钟执行时间（单位是滴答），如果执行失败则返回“-1”。

4. 应用实例——统计进程时间

例 7.20 是一个使用 times 和 sysconf 函数来测量一个可执行文件（进程）的执行时间进行测量的实例。

【例 7.20】统计进程的执行时间

应用代码使用 argv 来传递待执行的进程名称，在开始执行的时候记录一个时钟滴答数 start，然后在执行完退出之后再次记录一个时钟滴答数 end，通过这两个值来计算得到相应的执行时间，其流程如图 7.19 所示。在应用代码中定义了两个子函数 do_cmd 和 pr_times 分别用于执行该可执行文件（进程）和统计执行时间。

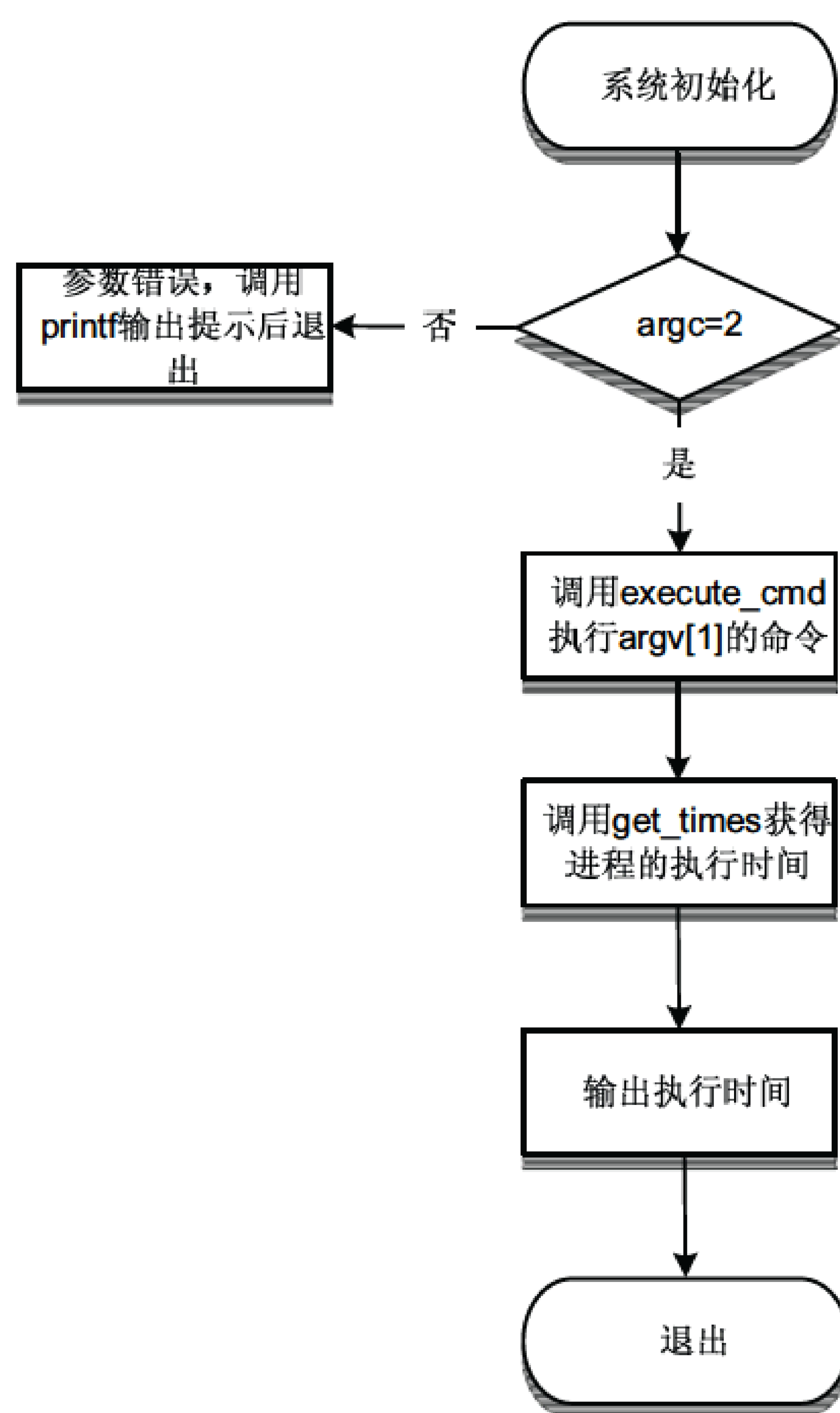


图 7.19　统计进程执行时间

实例的应用代码如下：

```
1    #include <sys/times.h>
2    #include <stdio.h>
3    #include <errno.h>
4    #include <stdlib.h>
5    #include <unistd.h>
6    //时间统计函数
7    static void get_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
8    {
9        static long                clktck = 0;
10       if (clktck == 0)                          //第一次获得时间
11       if ((clktck = sysconf(_SC_CLK_TCK)) < 0)
12       {
13           perror("调用 sysconf 函数错误.\n");
14       }
15       //以下为时间输出
16       printf("时钟时间:   %7.2f\n", real / (double) clktck);
17       printf("用户 CPU 时间:   %7.2f\n",(tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
18       printf("系统 CPU 时间:    %7.2f\n",(tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
19       printf("子进程时钟时间:   %7.2f\n",(tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
20       printf("子进程系统 CPU 时间:     %7.2f\n",(tmsend->tms_cstime - tmsstart->tms_cstime) / (double)
         clktck);
21   }
22
```

```
23    //执行并且对 cmd 命令计时
24    static void execute_cmd(char *cmd)
25    {
26        struct tms         tmsstart, tmsend;                 //时间结构体;
27        clock_t            start, end;                       //分别存放起始和停止时刻的时钟滴答数
28        int                status;                           //执行状态
29        printf("\n 当前执行的命令是: %s\n", cmd);          //输出对应的命令
30        if ((start = times(&tmsstart)) == -1)             //获得 start 时间
31        {
32            perror("调用 times 函数出错.\n");
33        }
34        if ((status = system(cmd)) < 0)                   //执行命令
35        {
36            perror("调用 system 函数出错\n");
37        }
38        if ((end = times(&tmsend)) == -1)                 //获得 end 时间
39        {
40              perror("调用 times 函数出错\n");
41        }
42        get_times(end-start, &tmsstart, &tmsend);         //计算运行时间
43    }
44    //主函数
45    int main(int argc, char *argv[])
46    {
47        int    i;
48        setbuf(stdout, NULL);                             //清空标准输出（屏幕）
49        if(argc != 2)                                     //参数数目错误
50        {
51           printf("请输入正确的命令.\n");
52           exit(0);
53        }
54        else
55        {
56           execute_cmd(argv[1]);                          //执行命令
57           exit(0);
58        }
59    }
```

将文件保存为 exam720time.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam720time。

```
alloy@ubuntu:~/linuxc/chapter7$ gcc exam720time.c -o exam720time
```

调用可执行文件 exam720time 执行 date 命令，可以看到 date 命令对应的时间统计。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam720time date

当前执行的命令是: date
2014 年  02 月  27 日  星期四  10:50:58 CST
时钟时间:        0.04
用户 CPU 时间:        0.00
```

```
系统 CPU 时间:        0.00
子进程时钟时间:       0.00
子进程系统 CPU 时间:          0.00
```

再次调用 exam720time 执行例 7.16 中生成的可执行文件 exam716fullprocess，在出现对应提示符“>”之后故意等待几秒才输入将要执行的命令 date，此时可以看到时钟时间为 3.01（对比前面为 0.04），这是 exam720time 进程的运行时间。

```
alloy@ubuntu:~/linuxc/chapter7$ ./exam720time ./exam716fullprocess

当前执行的命令是: ./exam716fullprocess
>date
2014 年 02 月 27 日 星期四 10:51:29 CST
子进程返回 0
时钟时间:        3.01
用户 CPU 时间:        0.00
系统 CPU 时间:        0.00
子进程时钟时间:       0.00
子进程系统 CPU 时间:          0.00
```

7.7 本章习题

1. 编写一个程序，使用 getpid 函数来获取当前进程的进程 ID。
2. 编写一个程序，使用 fork 函数来创建一个子进程，并分别输出父、子进程的进程 ID。
3. 编写一个程序，使用 fork 函数来创建一个子进程，并且说明父进程和子进程的随机返回问题。
4. 编写一个程序，使用 vfork 函数来创建一个子进程，并且说明父进程和子进程的随机返回问题。
5. 编写一个程序，考察 exit 函数的使用方法，在程序尚未运行到最后时使用 exit 函数退出，查看后面的程序语句是否会被执行？
6. 编写一个程序，考察_exit 函数的使用方法，在程序运行到最后时使用_exit 函数退出，查看程序的执行结果会发生怎样的变化。
7. 编写一个程序，使用 getpgrp 函数来获取当前进程的进程组 ID 并且输出。

第 8 章　Linux 的信号

信号（Signal）是一种软件中断，在第 3 章的例 3.14 中使用 Ctrl+C 快捷键退出运行的实例，其实质上就是使用了信号；信号在 Linux 操作系统中提供了一种处理异步事件的方法，可以很好地在多个进程之间进行同步和简单的数据交互，本章将介绍其使用方法，涉及以下内容：

- 信号的工作机制以及分类。
- 如何在 Linux 下使用信号。
- 信号集的基础知识及其使用方法。

8.1　Linux 的信号机制

信号机制是一种使用信号来进行进程之间传递消息的方法，其中信号的全称为软中断信号，简称软中断。

软中断信号（Signal，又简称为信号）用来通知进程发生了异步事件，进程之间可以互相通过系统调用 kill 函数来发送软中断信号，而 Linux 内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。



注　意

信号只是用来通知某进程发生了什么事件，但并不给该进程传递任何数据。

8.1.1　信号的工作方式

每个信号都有一个名字，这些名字都以三个字符 SIG 开头，在头文件<signal.h>中，这些信号都被定义为正整数，称为信号编号。



注　意

没有编号为 0 的信号，kill 函数对编号 0 有特殊的应用，在 POSIX.1 规范中，将此种信号编号值称为空信号。

Linux 内核支持 64 种不同的信号（具体内容将在第 8.1.2 小节中进行详细介绍），这些信号中的大部分都有了预先定义好的意义，但是都支持自定义动作，并且还提供了类似 SIGUSR1 这样由应用程序来定义的信号。

1. 信号的产生

Linux 中的信号可以由以下几种方式产生：

- 当用户按下某些终端按键之后引发终端产生的信号，例如在程序运行中按下“Ctrl+\”组合键将终止程序的运行。
- 硬件产生的一个异常信号，例如除数为 0、无效的内存引用等，这种异常信号通常会由硬件检测得到并将其通知 Linux 内核，然后内核为该条件发生时正在运行的进程产生适当的信号。
- 进程调用系统调用 kill 函数可以给一个进程或者进程组发送一个信号，需要注意的是此时发送和接收信号的进程/进程组的所有者必须相同。
- 用户也可以调用 kill 命令将信号发送给其他进程。
- 当检测到某种软件条件已经发生，并应将其通知有关进程的时候也会产生一个信号，例如 SIGURG 信号就是在接收到一个通过网络传送的外部数据时产生的。

2. 信号的处理方式

Linux 的每一个信号都有一个缺省的动作，典型的缺省动作是终止进程，当一个信号到来的时候收到这个信号的进程会根据信号的具体情况提供以下三种不同的处理方式：

- 类似中断的处理程序，对于需要处理的信号，进程可以指定处理函数，由该函数来处理。
- 忽略某个信号，对该信号不做任何处理，就像从未发生过一样。
- 对该信号的处理保留系统的默认值，这种缺省操作大多数是使得进程终止。进程通过系统调用 signal 函数来指定进程对某个信号的处理行为。

3. 信号的缺陷

作为一种进程交互机制，信号有一些局限性：

- 信号的系统开销太大。
- 发送信号的进程要进行系统调用。
- 内核要中断接收信号的进程，而且要管理它的堆栈，同时还要调用处理程序，之后恢复执行被中断的进程。
- 信号的数量非常有限，因为只存在有限的不同信号。
- 信号能传送的信息量十分有限，用户产生的信号不可能发送附加信息及各种参数。

所以，在实际使用中，信号机制常常用于进程之间的事件通知，而不应用于复杂的交互操作。

4. 信号的执行过程

一个典型的信号的执行过程是使用“Ctrl+C”组合键来中断一个进程的运行，对其操作部分说明如下，需要注意的是只有在前台运行的进程才能接收到“Ctrl+C”组合键的输入：

01 用户输入命令，在 Shell 下启动一个前台进程。

02 用户按下“Ctrl+C”，这个键盘输入产生一个硬件中断。

03 如果处理器当前正在执行这个进程的代码，则该进程的用户空间代码暂停执行，CPU 从用户态切换到内核态处理硬件中断。

04 终端驱动程序将 Ctrl+C 解释成一个 SIGINT 信号，记在该进程的 PCB 中（也可以说发送了一个 SIGINT 信号给该进程）。

05 当某个时刻要从内核返回到该进程的用户空间代码继续执行之前，首先处理 PCB 中记录的信号，发现有一个 SIGINT 信号待处理，而这个信号的默认处理动作是终止进程，所以直接终止进程，而不再返回它的用户空间代码。

Linux 内核给一个进程发送软中断信号的方法是：在进程所在的进程表项的信号域设置对应于该信号的位（内核通过在进程的 struct task_struct 结构中的信号域中设置相应的位来实现向一个进程发送信号）。

如果信号发送给一个正在睡眠的进程，那么要看该进程进入睡眠的优先级，如果进程睡眠在可被中断的优先级上，则唤醒进程；否则仅设置进程表中信号域相应的位，而不唤醒进程。这一点比较重要，因为进程检查是否收到信号的时机是：一个进程在即将从内核态返回到用户态时；或者，在一个进程要进入或离开一个适当的低调度优先级睡眠状态时。

内核处理一个进程收到的信号的时机是：在一个进程从内核态返回用户态时，所以，当一个进程在内核态下运行时，软中断信号并不立即起作用，要等到将返回用户态时才处理。进程只有处理完信号才会返回用户态，进程在用户态下不会有未处理完的信号。

内核处理一个进程收到的软中断信号是在该进程的上下文中，因此，进程必须处于运行状态。前面介绍概念的时候讲过，处理信号有三种类型：进程接收到信号后退出；进程忽略该信号；进程收到信号后执行用户自定义的使用系统调用 signal 注册的函数。当进程接收到一个它忽略的信号时，进程丢弃该信号，就像从来没有收到该信号一般而继续运行。如果进程收到一个要捕捉的信号，那么进程从内核态返回用户态时执行用户定义的函数，而且执行用户定义的函数方法很巧妙，内核是在用户栈上创建一个新的层，该层将返回地址的值设置成用户定义的处理函数的地址，这样进程从内核返回栈顶时就返回到用户定义的函数处，从函数返回再弹出栈顶时，才返回原先进入内核的地方。这样操作的原因是用户定义的处理函数不能且不允许在内核态下执行（如果用户定义的函数在内核态下运行的话，用户就可以获得任何权限）。

5. 信号处理的注意事项

在信号的处理方法中有几点特别要引起注意：

- 在一些系统中，当一个进程处理完中断信号返回用户态之前，内核清除用户区中设定的对该信号的处理例程的地址，即将下一次进程对该信号的处理方法改为默认值，除非在下一次信号到来之前再次使用 signal 系统调用。
- 如果要捕捉的信号发生于进程正在一个系统调用中时，并且该进程睡眠在可中断的优先级上，这时该信号引发进程调用 longjmp 函数跳出睡眠状态，返回用户态并执行信号处理例程；当从信号处理例程返回时，进程就像从系统调用返回一样，但返回了一个错误代码，指出该次系统调用曾经被中断。
- 若进程睡眠在可中断的优先级上，则当它收到一个要忽略的信号时，该进程被唤醒，但不做 longjmp 函数调用，一般是继续睡眠，一般情况下，用户感觉不到进程曾经被唤醒，而是像没有产生过信号一样。

- 内核对子进程终止（SIGCLD）信号的处理方法与其他信号有所区别。
- 如果一个进程调用 signal 系统，并设置了 SIGCLD 的处理方法，且该进程有子进程处于僵死状态，则内核将向该进程发送一个 SIGCLD 信号。

8.1.2 信号的分类和说明

参考第 8.1.1 小节中介绍的 Linux 下的信号源，可以把 Linux 下的信号分为如下几大类：

- 与进程终止相关的信号：当进程退出，或者子进程终止时，发出这类信号。
- 与进程例外事件相关的信号：如进程越界，或企图写入一个只读的内存区域（如程序正文区），或执行一个特权指令及其他各种硬件错误。
- 与在系统调用期间遇到不可恢复条件相关的信号：如执行系统调用 exec 时，原有资源已经释放，而目前系统资源又已经耗尽。
- 与执行系统调用时遇到非预测错误条件相关的信号：如执行一个并不存在的系统调用。
- 在用户态下的进程发出的信号：如进程调用系统调用 kill 向其他进程发送信号。
- 与终端交互相关的信号：如用户关闭一个终端，或按下“Break”键等情况。
- 跟踪进程执行的信号。

在 Linux 下可以使用“kill–l”命令来查看当前系统支持的全部信号，通常来说有 64 个默认的信号，其输出如图 8.1 所示（基于 Ubuntu 12.04 LTS 发行版）。

```
alloy@ubuntu:~/linuxc/chapter8$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

图 8.1 使用 kill 命令查看 Linux 中支持的信号

表 8.1 给出了信号编号为 1~31 的详尽介绍。

表 8.1 Linux 中的信号说明

信号名称	信号编号	信号说明
SIGHUP	1	在用户终端连接（正常或非正常）结束时发出，通常是在终端的控制进程结束时，通知同一会话期（Session）内的各个作业，这时它们与控制终端不再关联。在登录 Linux 系统的时候，系统会自动分配给登录用户一个控制终端。在这个终端运行的所有程序，包括前台进程组和后台进程组，一般都属于同一个会话。当用户退出 Linux 登录时，前台进程组和后台有对终端输出的进程将会收到 SIGHUP 信号。这个信号的默认操作为终止进程，因此前台进程组和后台有终端输出的进程就会中止。此外，对于与终端脱离关系的守护进程，这个信号用于通知它重新读取配置文件

（续表）

信号名称	信号编号	信号说明
SIGINT	2	程序终止（或中断，interrupt）信号，在用户键入 INTR 字符（通常是 Ctrl+C 或 Delete 键）时发出，用于通知前台进程组终止进程
SIGQUIT	3	和 SIGINT 信号类似，但由 QUIT 字符（通常是"Ctrl+\"）来控制，进程在收到 SIGQUIT 退出时会产生 core 文件，在这个意义上类似于一个程序错误信号
SIGILL	4	进程执行了非法指令，通常是因为可执行文件本身出现错误，或者试图执行数据段，堆栈溢出时也有可能产生这个信号
SIGTRAP	5	由断点指令或其他陷进（trap）指令产生，由调试器（debugger）使用，例如跟踪陷进信号
SIGABRT	6	调用 abort 函数时产生的信号，将会使进程非正常结束
SIGBUS	7	非法地址，包括内存地址对齐（alignment）出错，例如访问一个 4 个字长的整数，但其地址不是 4 的倍数。它与 SIGSEGV 的区别在于后者是由于对合法存储地址的非法访问触发的（如访问不属于自己存储空间或只读存储空间）
SIGFPE	8	在发生致命的算术运算错误时发出，不仅包括浮点运算错误，还包括溢出及除数为 0 等其他所有的算术错误
SIGKILL	9	用来立即结束程序的运行，本信号不能被阻塞、处理和忽略，如果管理员发现某个进程终止不了，可尝试发送这个信号
SIGUSR1	10	用户保留信号
SIGSEGV	11	试图访问未分配给登录用户的内存区，或试图向没有写权限的内存地址写数据
SIGUSR2	12	用户保留信号
SIGPIPE	13	管道破裂信号，当对一个读进程已经运行结束的管道执行写操作时产生。这种情况通常发生在进程间通信时，例如采用管道（FIFO）通信的两个进程，读管道还没有打开或者意外终止就向管道写时，写进程会收到 SIGPIPE 信号。此外，例如使用套接字（Socket）通信的两个进程，写进程在写 Socket 的时候，读进程已经终止
SIGALRM	14	时钟定时信号计算的是实际的时间或时钟时间，由 alarm 函数设定的时间段终止时，会产生该信号
SIGTERM	15	程序结束（terminate）信号，与 SIGKILL 不同的是该信号可以被阻塞和处理。通常用来要求程序自己正常退出，"kill"命令默认产生这个信号。如果进程终止不了，才会尝试 SIGKILL
SIGSTKFLT	16	堆栈错误
SIGCHLD	17	子进程结束时，父进程会收到这个信号。如果父进程没有处理这个信号，也没有等待子进程，则子进程虽然终止，但是还会在内核进程表中占有表项，这时的子进程称为僵尸进程，这种情况应该尽量避免，也就是说，父进程或者忽略该信号，或者捕捉它，或者等待它派生的子进程，或者父进程先终止，这时子进程的终止自动由 init 进程来接管

（续表）

信号名称	信号编号	信号说明
SIGCONT	18	让一个停止（stopped）的进程继续执行，此信号不能被阻塞，可以用一个信号处理程序来让程序在由停止状态变为继续执行时完成特定的工作。例如，重新显示提示符
SIGSTOP	19	停止（stopped）进程的执行，注意它和 terminate 以及 interrupt 的区别：该进程还未结束，只是暂停执行。此信号不能被阻塞、处理或忽略
SIGTSTP	20	停止进程的运行，但该信号可以被处理和忽略，用户键入 SUSP 字符时（通常是 Ctrl+Z）发出这个信号
SIGTTIN	21	当后台作业要从用户终端读取数据时，该作业中的所有进程会收到该信号，缺省时这些进程会停止执行
SIGTTOU	22	类似于 SIGTTIN，但在写终端（或修改终端模式）时收到
SIGURG	23	在套接字上出现类似紧急数据等情况时产生此信号
SIGXCPU	24	超过处理器源限制时产生的信号，可以使用 getrlimit/setrlimit 来读取或者改变该限制
SIGXFSZ	25	当进程企图扩大文件以至于超过文件大小资源限制时产生此信号
SIGVTALRM	26	虚拟时钟信号，类似于 SIGALRM，是计算的是该进程占用的处理器时间
SIGPROF	27	类似于 SIGALRM/SIGVTALRM，包括该进程使用的处理器时间以及系统调用的时间
SIGWINCH	28	当窗口大小发生改变时发出的信号
SIGIO	29	文件描述符准备就绪时发出的信号，此时可以开始输入或者输出操作
SIGPWR	30	硬件系统供电电源出现故障
SIGSYS	31	使用了一个非法的系统调用



注 意

在实际系统中，为了使管理员能在任何情况下都使用 SIGKILL（9）和 SIGSTOP（19）来结束或者停止一个指定的进程，所以这两个信号不能被应用程序捕捉和忽略。

在图 8.1 中编号为 1~31 的信号为传统 Linux 内核所支持的信号，是不可靠信号（非实时的），编号为 34 ~ 63 的信号是后来扩充的，称为可靠信号（实时信号）。不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号的丢失，而后者不会。

此外在不可靠信号中有 4 个比较特殊的信号，对其说明如下，其中前两个信号是不能被忽略的，而后两个信号是用户自定义的。

- SIGSTOP（19）：这个信号将中断进程的执行，对应键盘输入为“Ctrl+C”。
- SIGKILL（9）：这个信号将强制进程退出，对应键盘输入为“Ctrl+\”。
- SIGUSR1（12）和 SIGUSR2（12）：两个用户自定义信号。

除了使用“kill-l”命令之外，在 Linux 的帮助手册中对信号也有详细的定义，可以使用 man 7 signal 命令来查看更为详尽的说明，打开帮助文档之后下翻，可以看到如图 8.2 所示的分类详细说明，其中第 1 列是信号名称，第 2 列对应的是信号编号，第 3 列是对信号的处理方式，而第 4 列则

是对信号的详细说明。

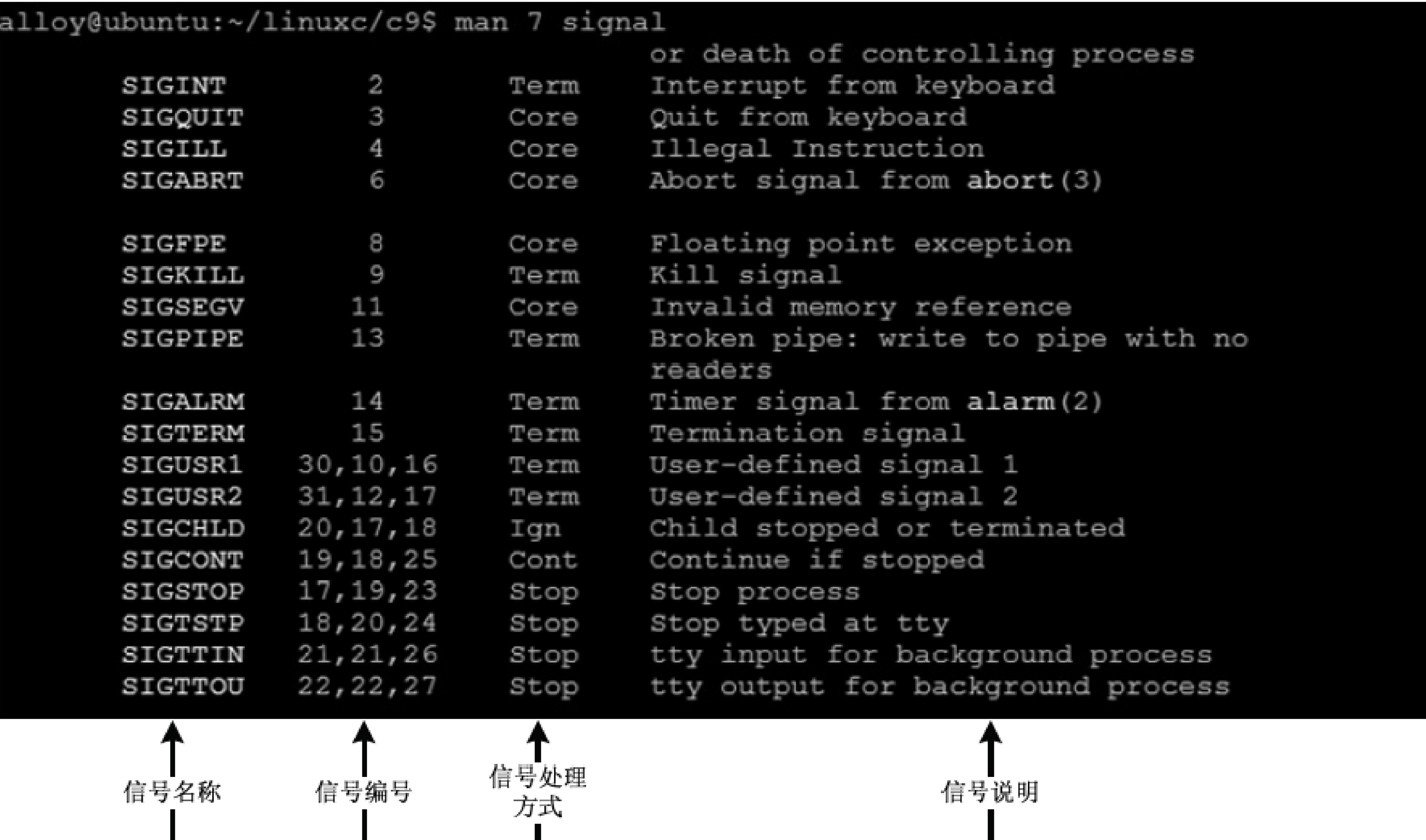


图 8.2　man 手册中的信号说明

在图 8.2 中对于信号的处理方式有 Core、Term、Ign、Cont 和 Stop 共 5 种，对它们的详细说明如下。

- Core：终止当前进程并且使用 Core Dump 保存进程用户空间的内存数据到 core 文件。
- Term：终止当前进程。
- Ign：忽略该信号。
- Stop：停止当前进程。
- Cont：继续执行之前停止的进程。

表 8.2 是 Linux 操作系统对信号处理方式的总结。

表 8.2　Linux 对信号的处理方式

信号的处理方式	信号列表
不能恢复至默认动作	SIGILL、SIGTRAP
默认会导致进程失败	SIGABRT、SIGBUS、SIGFPE、SIGILL、SIGIOT、SIGQUIT、SIGSEGV、SIGTRAP、SIGXCPU、SIGXFSZ
默认会导致进程退出的信号	SIGALRM、SIGHUP、SIGINT、SIGKILL、SIGPIPE、SIGPOLL、SIGPROF、SIGSYS、SIGTERM、SIGUSR1、SIGUSR2、SIGVTALRM
默认会导致进程停止	SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU
默认进程忽略的信号	SIGCHLD、SIGPWR、SIGURG、SIGWINCH

8.2　Linux 信号的使用方法

本节将介绍如何在 Linux 下使用信号机制，包括注册信号、发送信号、定时信号、退出信号等。

8.2.1 注册信号

对于已经有自己的功能动作的信号而言，其注册就是用一个用户自己定义的功能动作去替换 Linux 内核预定义的功能动作的操作，例如功能键“Ctrl+C”是中止当前进程的运行，当其被按下的时候当前进程会接收到一个 SIGINT 信号，然后对应该信号的操作是终止当前进程。用户可以使用信号注册将 SIGINT 信号对应的操作定义为用户期望的操作，例如在标准输出上输出一个字符串，在注册完成之后如果进程再次检测到 SIGINT 信号，即可输出字符串操作而不是终止退出。

Linux 提供了 singal 和 sigaction 函数用于信号的注册。

1. singal 函数

要对一个信号进行处理，就需要给出此信号发生时系统所调用的处理函数，singal 函数可以为一个特定的信号（除去无法捕捉的 SIGKILL 和 SIGSTOP 信号）注册相应的处理函数。如果正在运行的程序源代码里注册了针对某一特定信号的处理程序，不论当时程序执行到何处，一旦进程接收到该信号，相应的调用就会发生，对其标准调用格式说明如下：

```
#include <signal.h>
void (*signal (int signum, void (*handler) (int) ) ) (int);
```

对更为简洁的 signal 函数调用格式说明如下：

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

参数 signum 表示所注册函数针对的信号，其可以使用的值为 8.1.4 小节中介绍的信号关键字或者对应的编码（不推荐使用编码）；参数 handler 通常是指向调用函数的函数指针，这个函数是进程接收到信号之后的动作，即所谓的信号处理函数。

信号处理函数 handler 可能是用户自定义的一个函数，也可能是两个在 singal.h 头文件中进行定义的值：

- SIG_IGN：忽略 signum 指出的信号。
- SIG_DFL：调用系统定义的缺省信号处理。


注 意

信号处理函数的参数是要处理的信号值，并且不能为 SIGKILL 和 SIGSTOP 设置信号处理函数。

当 signal 函数调用成功之后，其返回信号以前的处理配置，如果调用失败则返回 SIG_ERR(-1)。

当程序执行 signal 后，表示从这个时候开始由 signum 指定的信号对应的操作是 handler 所传递的函数，需要注意的是并非是程序执行到 signal 这一行就立即对该信号进行操作，因为信号的产生是无法预期的，程序设计人员根本没法预知该在哪一行捕捉突如其来的信号。利用 signal 设置信号处理函数只是告诉系统对这个信号用什么程序来处理。

图 8.3 是一个 Linux 操作系统处理 signal 的注册信号过程。

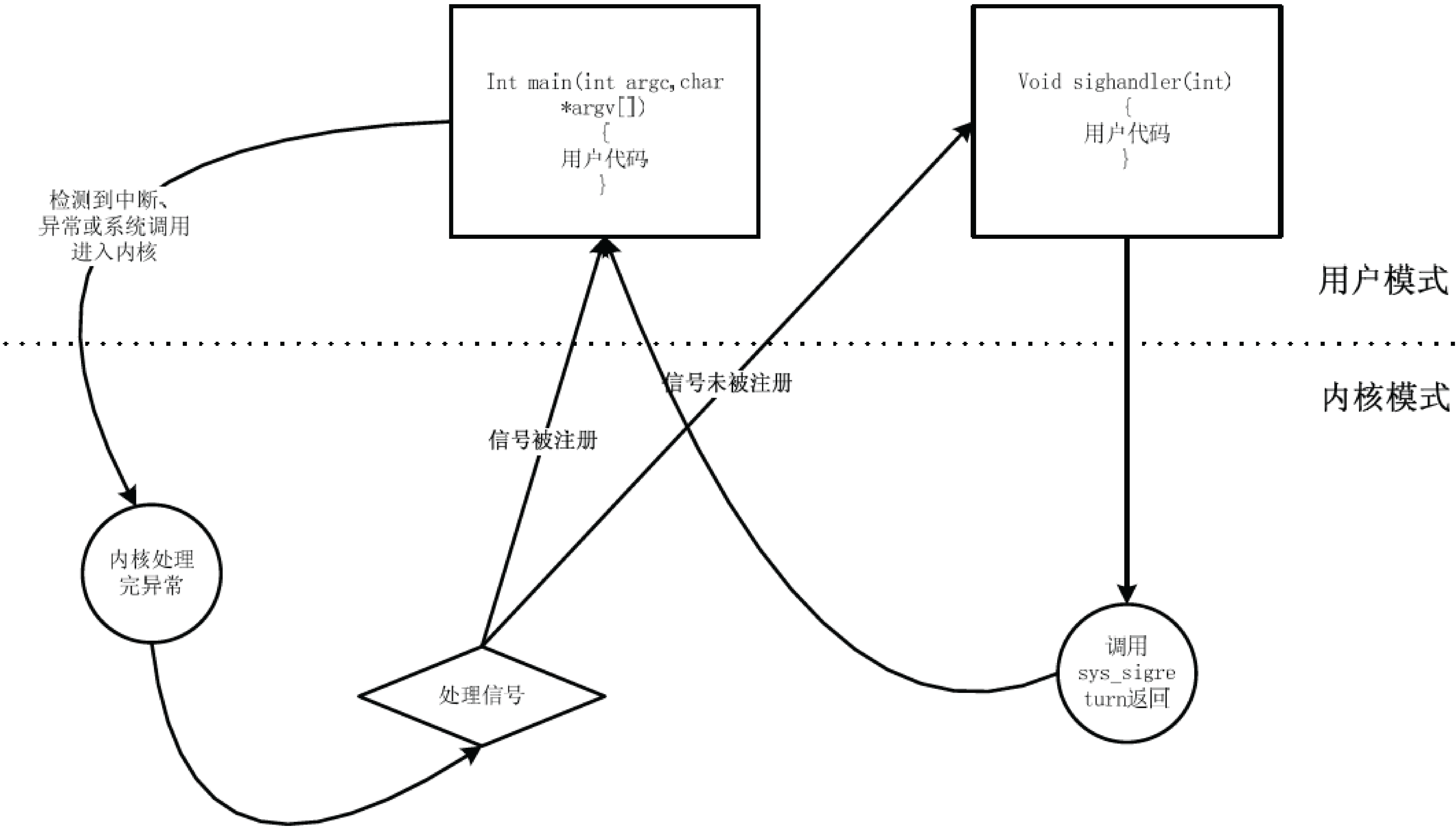


图 8.3　Linux 操作系统处理 singal 的注册信号

例 8.1 是一个使用 singal 函数对信号进行注册的实例。

【例 8.1】使用 signal 函数注册信号

应用代码使用 signal 函数来注册 signalDeal 函数，并作为 SIGINT 和 SIGQUIT 信号的处理函数，在该 signalDeal 函数中调用 printf 函数，用于在屏幕上输出当前进程接收到的信号对应的编号，其流程如图 8.4 所示。

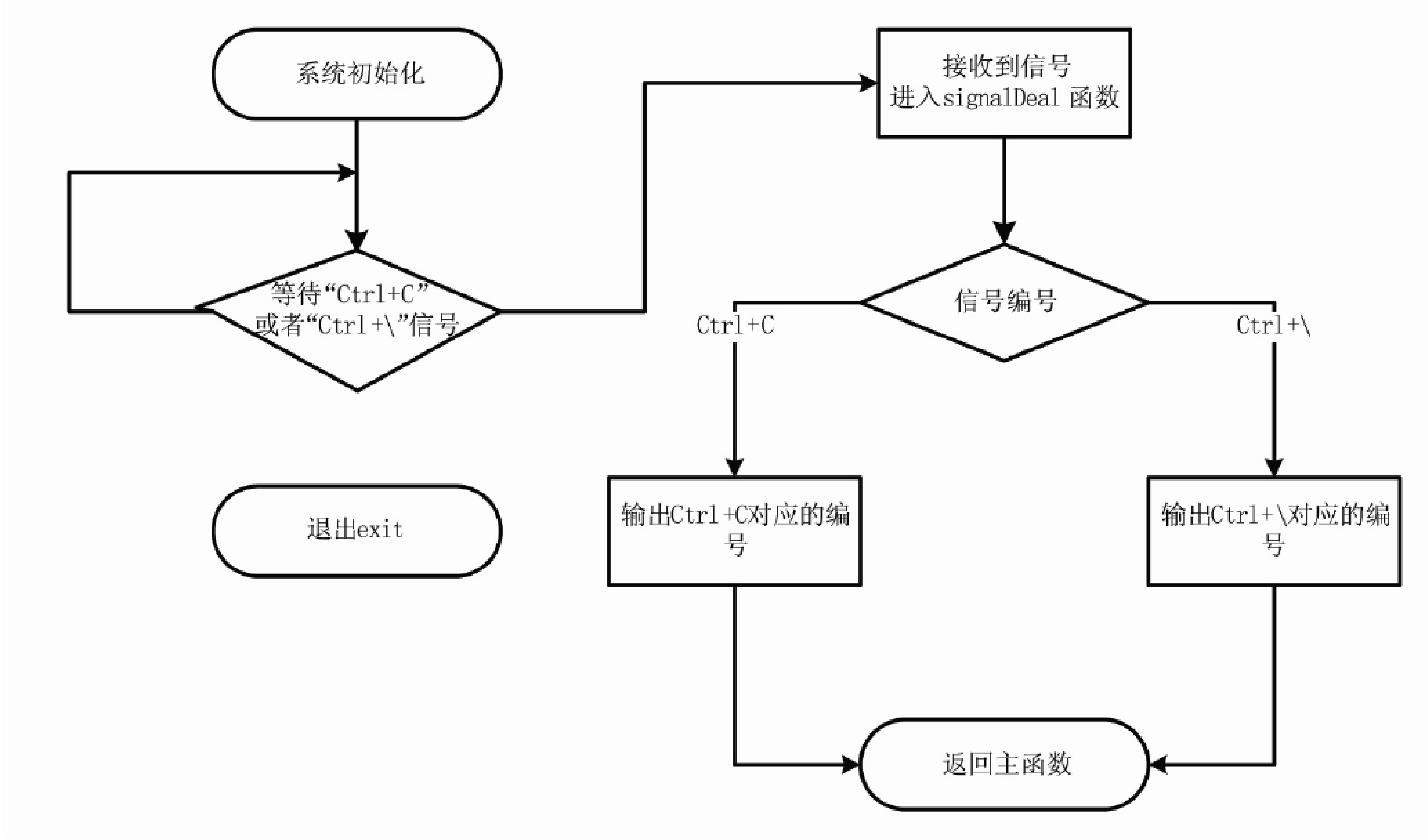


图 8.4　使用 signal 函数注册信号

实例的应用代码如下：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  //这是信号处理函数
5  void signalDeal(int sig)
6  {
7     if(sig == SIGINT)        //对应 Ctrl+C
8     {
9        printf("Ctrl+C 按键被按下。\n");
10    }
11    else if(sig == SIGQUIT)   //对应“Ctrl+/”
12    {
13       printf("Ctrl+/按键被按下.\n");
14    }
15    else
16    {
17       printf("其他信号。\n");
18    }
19 }
20 //以下是主函数
21 int main(int argc,char *argv[])
22 {
23    signal(SIGINT,signalDeal);     //注册 SIGINT 对应的处理函数
24    signal(SIGQUIT,signalDeal);    //注册 SIGQUIT 对应的处理函数
25    while(1)       //永远循环
26    {
27    }
28    return 0;
29 }
```

将文件保存为 exam801singal.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam801singal。

```
alloy@ubuntu:~/linuxc/chapter8$ gcc exam801singal.c -o exam801singal
```

运行该可执行文件，然后分别按下“Ctrl+C”和“Ctrl+\”，可以看到对应的输出，此时可以使用“Ctrl+Z”来发送一个 SIGTSTP 信号退出当前正在执行的进程。

```
alloy@ubuntu:~/linuxc/chapter8$ ./exam801singal
^C Ctrl+C 按键被按下。
^\ CtrL+/按键被按下.
^C Ctrl+C 按键被按下。
^Z
[1]+   已停止                    ./exam801singal
```

2. sigaction 函数

如果觉得 signal 函数功能不够强大，可以使用功能更加强大的 sigaction 函数来完成相应的注

册工作，对其标准调用格式说明如下：

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,struct sigaction *oldact);
```

其中，参数 signum 指定要处理的信号（除 SIGKILL 和 SIGSTOP 之外），act 和 oldact 都是指向信号动作结构的指针，该结构的定义如下：

```
struct sigaction
{
        void (*sa_handler)(int);
        void(*sa_sigaction)(int,siginfo_t *,void *);
        sigset_t sa_mask;
        int sa_flags;
}
```

其中 sa_handler 用于指向信号处理函数的地址。参数 sa_sigaction 是指向函数的指针。它指向的函数有三个参数，其中第二个为 siginfo_t 结构体，定义如下：

```
struct siginfo_t
{
        int si_signo;           /*Signal number */
        int si_errno;           /*An errno value */
        int si_code;            /*Signal code */
        pid_t si_pid;           /*Sending process ID */
        uid_t si_uid;           /*Real user ID of sending process */
        int si_status;          /*Exit value or signal */
        clock_t si_utime;       /*User Time consumed */
        clock_t si_stime;       /*System time consumed */
        signal_t si_value;      /*Signal value */
        int si_int;             /* POSIX.1b signal */
        void *si_ptr;           /*POSIX.1b signal */
        void *si_addr;          /*Memory location that caused fault */
        int si_band;            /*Band event */
        int si_fd;              /*File descriptor */
}
```

sa_flags 用于指示信号处理函数的不同选项，具体的可选参数如表 8.3 所示。可以通过位运算的或运算（OR）连接不同的参数，从而实现所需的选项设置，将其赋值为 0 则选用所有的默认选项。

表 8.3　sa_flags 可选标志及对应设置

sa_flags	对应设置
SA_NOCLDSTOP	用于指定信号 SIGCHLD，当子进程被中断时，不产生此信号，当且仅当子进程结束时产生该信号
SA_NOCLDWAIT	当信号为 SIGCHLD 时，此选项可以避免子进程的僵死
SA_NODEFER	当信号处理程序正在运行时，不阻塞信号处理函数自身的信号功能

（续表）

sa_flags	对应设置
SA_NOMASK	同 SA_NODEFER
SA_ONESHOT	当用户注册的信号处理函数被调用过一次之后，该信号的处理程序恢复为缺省的处理函数
SA_RESETHAND	同 SA_ONESHOT
SA_RESTART	使本来不能进行自动重新运行的系统调用自动重新启动
SA_SIGINFO	表明信号处理函数是由 sa_sigaction 指定，而不是由 sa_handler 指定，它将显示更多处理函数的信息

例 8.2 是使用 sigaction 函数注册信号的实例。

【例 8.2】使用 sigaction 函数注册信号

应用代码使用 sigaction 函数来实现与例 8.1 完全相同的功能，需要注意的是其中使用了和信号集相关的函数 sigemptyset 来清空信号集中的所有信号，其对应的流程可以参考图 8.4。

实例的应用代码如下：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  //这是使用 sigaction 函数注册的信号函数
5  void signalDeal(int sig,siginfo_t *info,void *t)
6  {
7     if(sig == SIGINT)       //对应 Ctrl+C
8     {
9        printf("Ctrl+C 按键被按下。\n");
10    }
11    else if(sig == SIGQUIT)    //对应“Ctrl+/”
12    {
13       printf("Ctrl+/按键被按下.\n");
14    }
15    else
16    {
17       printf("其他信号。\n");
18    }
19 }
20 int main(int argc,char *argv[])
21 {
22    struct sigaction act;                //定义 sigaction 结构体
23    act.sa_sigaction = signalDeal;       //指定信号处理函数
24    sigemptyset(&act.sa_mask);           //清空信号集中的信号
25    act.sa_flags = SA_SIGINFO;           //信号附带的参数可以被传递到处理函数中
26    sigaction(SIGINT,&act,NULL);         //设置 SIGINT 处理函数
```

```
27      sigaction(SIGQUIT,&act,NULL);       //设置 SIGQUIT 处理函数
28      while(1)
29      {
30      }
31      return 0;
32  }
```

将文件保存为 exam802sigaction.c，然后在终端中使用 gcc 进行编译链接，生成可执行文件 exam802sigaction。

```
alloy@ubuntu:~/linuxc/chapter8$ gcc exam802sigaction.c -o exam802sigaction
```

运行该可执行文件，可以看到与例 8.1 相同的输出，同样可以使用 Ctrl+Z 快捷键退出正在运行的进程。

```
alloy@ubuntu:~/linuxc/chapter8$ ./exam802sigaction
^C Ctrl+C 按键被按下。
^\ Ctrl+/按键被按下.
^\ Ctrl+/按键被按下.
^Z
[2]+  已停止                    ./exam802sigaction
```

8.2.2　发送信号

在 Linux 中用户进程可以调用 kill 函数和 raise 函数来完成相应的信号发送操作，前者用于给其他进程发送信号，而后者用于给进程自身发送信号。

1. kill 函数

kill 函数将信号发送给进程或者进程组，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

参数 pid 表示 kill 函数发送信号对象的进程或进程组号，其取值说明如表 8.4 所示；参数 sig 为需要发送的信号编码，当该函数调用成功时，其返回值为“0”，如果调用失败则其返回值为“−1”。

表 8.4　kill 函数的 pid 参数

pid	含义
pid>0	将信号发送给进程号为 pid 的进程
pid=0	将信号发送给与目前进程相同进程组的所有进程
pid<0&& pid!=−1	向进程组 ID 为 pid 绝对值的进程组中的所有进程发送信号
pid=−1	除发送进程自身外，还向所有进程 ID 大于 1 的进程发送信号

需要注意的是：进程使用 kill 函数向另外一个进程发送信号需要相应的权限，超级用户则可以将信号发送给任意进程，非超级用户需要发送者和接收者的实际或者有效用户 ID 必须相同。



SIGCONT 信号除外，任何进程都可以将该信号发送给属于同一会话的任何其他进程。

另外一个需要注意的问题是：sig 参数对应的是信号编码值，当其为 0 时（即空信号），实际不发送任何信号，但照常进行错误检查，因此，可用于检查目标进程是否存在，以及当前进程是否具有向目标发送信号的权限（root 权限的进程可以向任何进程发送信号，非 root 权限的进程只能向属于同一个 session（会话）或者同一个用户的进程发送信号）。

2. raise 函数

如果当前进程需要向自身发送一个信号，可以使用 raise 函数，对其标准调用格式说明如下：

```
#include <signal.h>
int raise(int sig);
```

sig 参数是需要向自身发送信号的信号编码，如果函数调用成功，则返回“0”，如果出错，则返回“-1”。



其实 raise 函数其实等同于调用 kill(getpid(),signo)。

3. sigqueue 函数

sigqueue 主要是针对实时信号提出的(当然也支持前 32 种)信号发送函数，通常与函数 sigaction 配合使用，其标准调用格式如下：

```
#include <signal.h>
int sigqueue(pid_t pid, int sig, const union sigval value);
```

如果 sigqueue 函数调用成功，则返回 0，如果出错则返回-1，其第 1 个参数 pid 是指定接收信号的进程 ID，第 2 个参数 sig 用于确定即将发送的信号，第 3 个参数是一个联合数据结构 union sigval，指定了信号传递的参数，即通常所说的 4 字节值，其具体定义如下：

```
typedef union sigval
{
int sival_int;
void *sival_ptr;        //指向要传递的信号参数
}sigval_t;
```

sigqueue 比 kill 传递了更多的附加信息，但 sigqueue 只能向一个进程发送信号，而不能发送信号给一个进程组。如果 sig=0，将会执行错误检查，但实际上不发送任何信号，0 值信号可用于检查 pid 的有效性及当前进程是否有权限向目标进程发送信号。

在调用 sigqueue 时，sigval_t 指定的信息会拷贝到 3 参数信号处理函数（3 参数信号处理函数指的是信号处理函数由 sigaction 安装，并设定了 sa_sigaction 指针，在稍后实例中读者可以清楚地看到）的 siginfo_t 结构中，信号处理函数就可以处理这些信息了。由于 sigqueue 系统调用支持发

送带参数的信号，所以比 kill 系统调用的功能要灵活和强大得多。

4. 发送信号的应用实例

例 8.3~例 8.6 是几个使用 kill、raise 和 sigqueue 函数发送信号的应用实例。

【例 8.3】使用 raise 函数发送信号

应用代码调用了 raise 函数向自身进程发送一个 SIGABRT 信号，当发送失败时会返回一个为“-1”的值，此时进程会在屏幕上输出字符串来提示调用 raise 函数失败，同时也会提示进程没有收到对应的 SIGABRT 信号，如果进程接收到了该信号，则将自动中止，直接退出。

实例的应用代码如下：

```
1  #include <stdio.h>
2  #include <signal.h>
3  #include <stdlib.h>
4  int main(int argc,char *argv[])
5  {
6      printf("这是一个 raise 函数的应用实例\n");
7      if(raise(SIGABRT) == -1)                    //向进程本身发送 SIGABRT 信号失败
8      {
9         printf("调用 raise 函数失败!\n");         //提示发送失败，然后退出
10        exit(1);
11     }
12     printf("raise 发送 SIGABRT 信号没有成功!\n");  //如果进程被自己中止则不显示
13     return 0;
14 }
```

将文件保存为 exam803raise.c，在终端调用 gcc 进行编译链接，生成可执行文件 exam803raise。

```
alloy@ubuntu:~/linuxc/chapter8$ gcc exam803raise.c -o exam803raise
```

在当前目录中执行该文件，可以看到发送 SIGABRT 信号成功，自身进程被中止退出。

```
alloy@ubuntu:~/linuxc/chapter8$ ./exam803raise
这是一个 raise 函数的应用实例
已放弃（核心已转储）
```

【例 8.4】使用 kill 函数发送信号

这是一个父进程调用 kill 函数向子进程发送 SIGABRT 信号，让子进程退出的实例，首先使用 fork 函数创建一个子进程，使子进程休眠 10 秒，然后在父进程中调用 kill 函数向子进程发送一个 SIGABRT 信号，子进程收到该信号后退出，其流程如图 8.5 所示。

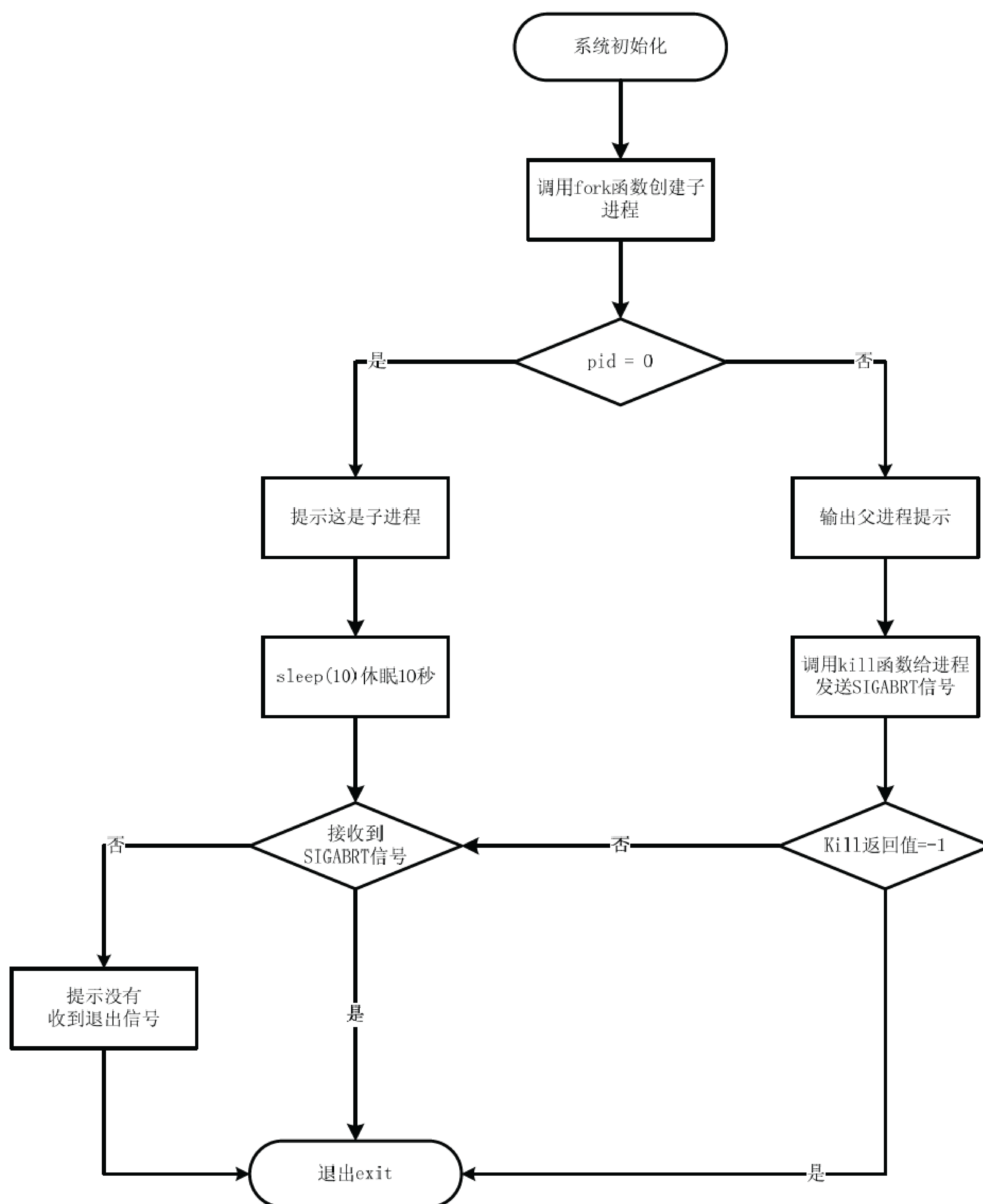


图 8.5　使用 kill 函数发送信号

实例的应用代码如下：

```
1  #include<signal.h>
2  #include<stdlib.h>
3  #include<stdio.h>
4  int main(int argc,char *argv[])
5  {
6     pid_t pid;
7     pid = fork();                    //创建子进程，进程 ID 存放在 pid 中
8     if(pid == 0)                     //子进程
9     {
```

```
10          printf("这是子进程!\n");
11          sleep(10);                                    //休眠 10 秒
12          printf("子进程没有收到退出指令!\n");          //如果接收到 SIGABRT 不会打印
13          return;
14       }
15       else                                             //这是父进程
16       {
17          printf("父进程调用 kill 函数向子进程%d 发送 SIGABRT 信号\n",pid);
18          sleep(1);                                     //休眠 1 秒
19          if(kill(pid ,SIGABRT) == -1)                  //如果调用 kill 函数失败
20          {
21             printf("调用 kill 函数失败!\n");
22          }
23       }
24       return 0;
25    }
```

将文件保存为 exam804kill.c，在终端使用 gcc 进行编译链接，生成 exam804kill 可执行文件。

```
alloy@ubuntu:~/linuxc/chapter8$ gcc exam804kill.c -o exam804kill
```

运行 exam804kill 可执行文件，可以看到父进程向子进程发送 SIGABRT 信号成功，子进程退出。

```
alloy@ubuntu:~/linuxc/chapter8$ ./exam804kill
父进程调用 kill 函数向子进程 3579 发送 SIGABRT 信号
这是子进程!
```

【例 8.5】使用 sigqueue 函数发送信号

这是一个使用 sigqueue 函数向进程自身发送 SIGUSR1（用户自定义信号 1）并且获取该信号的信号值的实例。

实例的应用代码如下：

```
1    /*
2    使用 sigqueue 函数向进程自身发送一个 SIGUSR1 信号，
3    并获取该信号的信号值
4    */
5    #include<stdio.h>
6    #include<signal.h>
7    #include<stdlib.h>
8    //SIGUSR1 的处理函数
9    void signalDeal(int signo,siginfo_t *info,void *context)
10   {
11         char *pMsg=(char*)info->si_value.sival_ptr;
12         printf("接收到的信号标号是:%d\n", signo);
13         printf("接收到信息:%s\n", pMsg);
14   }
15   //主函数
16   int main(int argc,char *argv[])
```

```
17  {
18          struct sigaction sigAct;
19          sigAct.sa_flags = SA_SIGINFO;
20          sigAct.sa_sigaction = signalDeal;
21          if(sigaction(SIGUSR1,&sigAct,NULL)==-1)
22          {
23              printf("sigaction 函数调用失败!\n");
24              exit(1);
25          }
26          sigval_t val;
27          char pMsg[ ]="this is a test!";            //这是一段测试用的字符串
28          val.sival_ptr = pMsg;
29          if(sigqueue(getpid(),SIGUSR1,val) == -1)
30          {
31              printf("sigqueue 函数调用失败!\n");
32              exit(1);
33          }
34          sleep(3);                                  //休眠 3 秒
35          return 0;
36  }
```

将文件保存为 exam805sigqueue.c，然后在终端编译生成可执行文件 exam805sigqueue。

```
alloy@ubuntu:~/linuxc/chapter8$ gcc exam805sigqueue.c -o exam805sigqueue
```

执行 exam805sigqueue，可以看到输出了 signalDeal 中的字符串。

```
alloy@ubuntu:~/linuxc/chapter8$ ./exam805sigqueue
接收到的信号标号是:10
接收到信息:this is a test!
```

例 8.3~例 8.5 是 3 个信号发送函数的基础实例，而例 8.6 和例 8.7 则是两个进程间实际应用的信号发送实例。

【例 8.6】进程间使用信号进行同步

例 8.6 是一个主进程和子进程使用信号进行同步的实例，首先使用 fork 函数创建一个子进程，然后在主进程中使用 sleep 函数每隔一秒调用 kill 函数向子进程发送一个用户自定义信号 SIGUSR1；子进程则使用 signal 函数对用户自定义信号 SIGUSR1 进行注册，使用 singalUSR1Deal 函数来对信号进行处理，在函数中输出了当前的时间信息，其流程如图 8.6 所示。

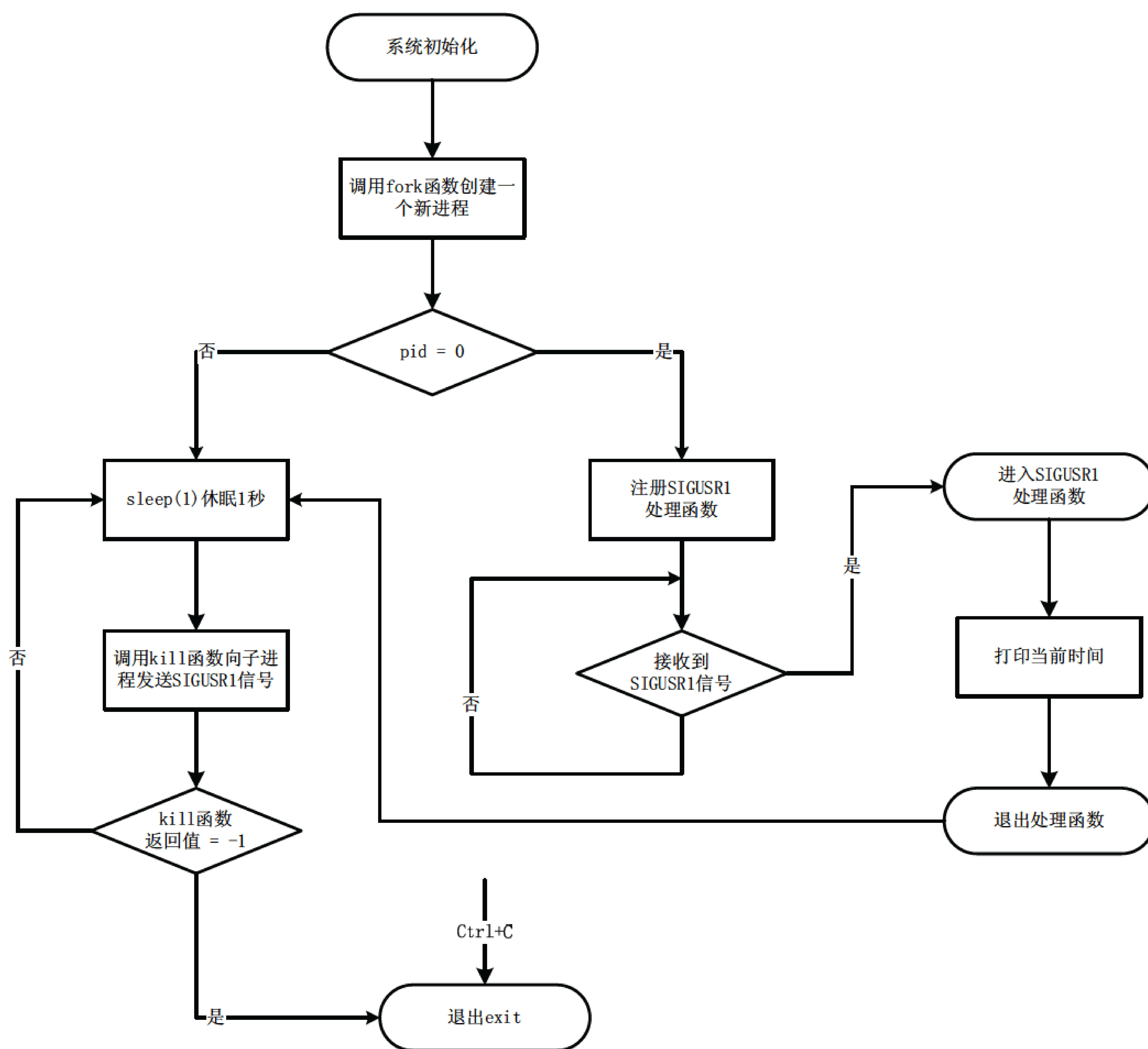


图 8.6　进程间使用信号进行同步

实例的应用代码如下：

```
1  /*
2  主进程休眠 1 秒，给子进程发送一个 usr1 信号,子进程接收到
3  usr1 信号后进入注册信号处理函数，在屏幕上输出当前时间。
4  */
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <signal.h>
8  #include <time.h>
9  //这是 USR1 的信号处理函数，用于在屏幕上输出时间信息
10 void singalUSR1Deal(int iSig)
11 {
12     time_t timetmp;                          //定义一个时间结构体变量
13     if(iSig == SIGUSR1)                      //如果是用户信号 1
14     {
```

```
15          time(&timetmp);                        //获得当前时间参数
16          printf("%s",ctime(&timetmp));          //在屏幕上输出当前时间
17      }
18      return;
19  }
20  //以下为主函数
21  int main(int argc,char *argv[])
22  {
23      pid_t pid;                                 //进程的 ID
24      pid = fork();                              //调用 fork 创建一个新的 ID
25      if(pid != 0)                               //主进程
26      {
27          while(1)                               //循环
28          {
29              sleep(1);                          //休眠 1 秒
30              if(kill(pid,SIGUSR1) == -1)        //调用 kill 函数向子进程发送 SIGUSR1 信号
31              {
32                  printf("向子进程发送 SIGUSR1 失败。\n");
33                  exit(0);                       //退出
34              }
35          }
36      }
37      else                                       //子进程
38      {
39          signal(SIGUSR1,singalUSR1Deal);        //注册 SIGUSR1
40          while(1)
41          {
42          }
43      }
44  }
```

将文件保存为 exam806killusr1.c，在终端使用 gcc 编译运行，生成可执行文件 exam806killusr1。

```
alloy@ubuntu:~/linuxc/chapter8$ gcc exam806killusr1.c -o exam806killusr1
```

执行 exam806killusr1 文件，可以看到子进程以秒为间隔在屏幕上输出对应的时间字符串，可以使用组合键“Ctrl+C”退出当前的运行。

```
alloy@ubuntu:~/linuxc/chapter8$ ./exam806killusr1
Fri Feb 28 23:35:16 2014
Fri Feb 28 23:35:17 2014
Fri Feb 28 23:35:18 2014
^C
```

【例 8.7】进程间使用信号进行同步

例 8.7 是一个多进程间使用信号进行同步的实例，其在例 8.6 的基础上增加了一个子进程。首先判断输入的文件参数是否正确，如果不正确则退出；如果正确则打开或者创建 argv+1 参数指定的文件。在主进程中创建了两个子进程，子进程 1 接收主进程发送的信号 SIGUSR1，然后在屏幕

上输出当前的时间信息；子进程 2 接收主进程发送的信号 SIGUSR2，然后将一个字符串写入主进程打开的文件；主进程调用 sleep 函数休眠，然后每隔 1 秒向子进程 1 和子进程 2 发送 SIGUSR1 和 SIGUSR2 信号。

实例的应用代码如下：

```
1   /*
2   主进程创建 2 个子进程，给子进程 1 每隔 1 秒发送一个 usr1 信号,子进程 1 接收到
3   usr1 信号后进入注册信号处理函数，在屏幕上输出当前时间；给子进程 2 每隔
4   1 秒发送一个 usr2 信号，子进程 2 接收到 usr2 信号后对一个在主进程中创建的文
5   件进行写入字符串操作
6   */
7   #include <stdlib.h>
8   #include <stdio.h>
9   #include <signal.h>
10  #include <time.h>
11  #include <fcntl.h>
12  #include <string.h>
13  #include <sys/types.h>
14  //这是 USR1 的信号处理函数，用于在屏幕上输出时间信息
15
16  #define TRUE 0x01
17  #define FALSE 0x00
18
19  unsigned char flg = FALSE;                    //标志位定义
20
21  void singalUSR1Deal(int iSig)
22  {
23    time_t timetmp;                             //定义一个时间结构体变量
24    if(iSig == SIGUSR1)                         //如果是用户信号 1
25    {
26      time(&timetmp);                           //获得当前时间参数
27      printf("%s",ctime(&timetmp));             //在屏幕上输出当前时间
28    }
29    return;
30  }
31  //这是 USR2 的信号处理函数，用于向一个文件中写入字符串
32  void singalUSR2Deal(int iSig)
33  {
34    if(iSig == SIGUSR2)                         //如果是用户信号 2
35    {
36      if(flg == FALSE)                          //如果标志为假
37      {
38        flg = TRUE;                             //修改标志位
39      }
40    }
41  }
42
43  //以下为主函数
```

```
44  int main(int argc,char *argv[])
45  {
46      pid_t pid1,pid2;                          //进程的 ID
47      int fd;                                   //文件描述符
48      char writebuf[] = "this is a test!\n";    //待写入字符串
49      int writecounter = 0;                     //用于记录写入的偏移量
50      int temp = 0,seektemp = 0,j = 0;          //都是用于计算文件偏移的临时变量
51      if (argc != 2)                            //如果参数不正确
52      {
53          printf("请输入正确的文件参数。\n");
54          return 0;
55      }
56      fd = open(*(argv+1),O_RDWR|O_CREATE,S_IRWXU); //打开或者创建一个文件
57      pid1 = fork();                            //调用 fork 创建一个新的进程
58      if(pid1 != 0)                             //主进程
59      {
60          pid2 = fork();                        //创建第 2 个子进程
61          if(pid2 != 0)                         //主进程
62          {
63              while(1)                          //循环
64              {
65                  sleep(1);                     //休眠 1 秒
66                  if(kill(pid1,SIGUSR1) == -1)  //调用 kill 函数向子进程 1 发送 SIGUSR1 信号
67                  {
68                      printf("向子进程 1 发送 SIGUSR1 失败。\n");
69                      exit(0);                  //退出
70                  }
71                  if(kill(pid2,SIGUSR2) == -1)  //调用 kill 向子进程 2 发送 SIGUSR2 信号
72                  {
73                      printf("向子进程 2 发送 SIGUSR2 失败。\n");
74                      exit(0);                  //退出
75                  }
76              }
77          }
78          else                                  //这是子进程 2 的操作
79          {
80              signal(SIGUSR2,singalUSR2Deal);   //注册 SIGUSR2 的处理函数
81              while(1)
82              {
83                  while(flg == FALSE);          //如果标志为假则等待
84                  flg = FALSE;                  //修改标志位
85                  printf("这是子进程 2.\n");     //屏幕输出提示
86                  if(writecounter == 0)         //第一次写入
87                  {
88                      temp = write(fd,writebuf,strlen(writebuf));       //写入数据
89                      seektemp = lseek(fd,0,SEEK_CUR);                  //获得当前偏移量
90                      writecounter++;
91                  }
92                  else
```

```
93              {
94                  j = strlen(writebuf)*writecounter;
95                  seektemp = lseek(fd,j,SEEK_SET);
96                  temp = write(fd,writebuf,strlen(writebuf));
97                  writecounter++;
98              }
99          }
100     }
101 }
102 else                                          //子进程
103 {
104     signal(SIGUSR1,singalUSR1Deal);           //注册 SIGUSR1 的处理函数
105     while(1)
106     {
107     }
108 }
109 }
```

将文件保存为 exam807killUSR.c，在终端使用 gcc 进行编译链接，生成可执行文件 exam807killUSR。

```
alloy@ubuntu:~/linuxc/chapter8$ gcc exam807killUSR.c -o exam807killUSR
```

执行 exam807killUSR 文件，对一个命名为 killUSRtest.txt 的文件进行操作，可以看到子进程 1 在屏幕上输出对应的时间信息，而子进程 2 输出“这是子进程 2”的字符串，可以使用“Ctrl+C”快捷键退出当前的进程操作。

```
alloy@ubuntu:~/linuxc/chapter8$ ./exam807killUSR killUSRtest.txt
这是子进程 2.
Sat Mar  1 11:03:30 2014
这是子进程 2.
Sat Mar  1 11:03:31 2014
这是子进程 2.
Sat Mar  1 11:03:32 2014
这是子进程 2.
Sat Mar  1 11:03:33 2014
这是子进程 2.
Sat Mar  1 11:03:34 2014
^C
```

使用“cat –n”命令查看 killUSRtest.txt 文件的内容，可以看到如下的输出。

```
alloy@ubuntu:~/linuxc/chapter8$ cat killUSRtest.txt -n
 1  this is a test!
 2  this is a test!
 3  this is a test!
 4  this is a test!
 5  this is a test!
```

8.2.3　定时信号

在 Linux 的应用程序中，常常需要每隔一段时间使线程执行一个动作，此时可以使用 SIGALRM 信号量，Linux 内核同样提供了相应的操作函数 alarm，对其标准调用格式说明如下：

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

参数 seconds 指定了下一次发送信号的时间，即在当前时间的 seconds 秒后，向进程本身发送 SIGALRM 信号，又称为闹钟时间。进程调用 alarm 后，任何以前的 alarm 调用都将无效。如果参数 seconds 为 0，那么进程内将不再包含任何闹钟时间。

如果调用 alarm 之前，进程中已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回 0。

例 8.8 是一个 alarm 函数的应用实例。

【例 8.8】使用 alarm 函数进行定时

利用 alarm 函数定时 3 秒，然后在对应的 SIGALRM 信号处理函数 signalDeal 中输出一个字符串提示。为了保证进程不至于在 alarm 规定的时间内已经退出，使用了 for 循环语句调用 sleep 函数定时 4 秒，并且在其中依次打印当前的定时时间长度。

实例的应用代码如下：

```
1   #include <unistd.h>
2   #include <signal.h>
3   #include <stdio.h>
4   //SIGALRM 的处理函数
5   void signalDeal(int sig)
6   {
7       if(sig == SIGALRM)
8       {
9           printf("这是定时信号的处理函数!\n");
10          return;
11      }
12  }
13  //这是主函数
14  int main(int argc,char *argv[])
15  {
16      int i = 0;
17      signal(SIGALRM,signalDeal);     //注册 SIGALRM 的处理函数
18      alarm(3);                       //3 秒定时
19      for(i=1;i<5;i++)
20      {
21          printf("sleeping %d ...\n",i);
22          sleep(1);
23      }
24      return 0;
25  }
```

将文件保存为 exam808alarm.c，在终端运行 gcc 进行编译链接，生成可执行文件 exam808alarm。

```
alloy@ubuntu:~/linuxc/chapter8$ gcc exam808alarm.c -o exam808alarm
```

运行该可执行文件，可以看到在主程序定时 3 秒后 alarm 定时到来，输出对应的字符串，然后主程序在 for 循环的定时第 4 秒后退出。

```
alloy@ubuntu:~/linuxc/chapter8$ ./exam808alarm
sleeping 1 ...
sleeping 2 ...
sleeping 3 ...
这是定时信号的处理函数!
sleeping 4 ...
```

8.2.4 退出信号

如果进程在执行过程中出现了异常，可以调用 abort 函数向进程发送 SIGABRT 信号使其退出，对 abort 函数的标准调用格式说明如下：

```
#include <stdlib.h>
void abort(void);
```

abort 函数用于将 SIGABRT（退出）信号发送给调用的进程，其没有返回值，例 8.9 是一个 abort 函数的应用实例。

【例 8.9】使用 abort 函数发送退出信号

这是一个向主进程自身发送 SIGABRT 信号来退出主进程的实例，其中 exit 函数的参数 EXIT_SUCCESS 表示这是一个成功的退出。

实例的应用代码如下：

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <signal.h>
4  int main(int argc,char *argv[])
5  {
6      abort();      //退出
7      exit(EXIT_SUCCESS);
8  }
```

将文件保存为 exam809abort.c，然后在终端使用 gcc 进行编译链接，生成可执行文件 exam809abort。

```
alloy@ubuntu:~/linuxc/chapter8$ gcc exam809abort.c -o exam809abort
```

运行该可执行文件，可以看到进程退出。

```
alloy@ubuntu:~/linuxc/chapter8$ ./exam809abort
已放弃 (核心已转储)
```

8.3　Linux 的信号集

在 Linux 系统的实际应用中，常常需要将多个信号组合起来使用，这种用来表示多个信号的数据类型被称为 Linux 的信号集（signal set），其定义格式为 sigset_t。

信号集的数据格式定义结构位于 signal.h 头文件中，对其说明如下：

```
typedef struct
{
unsigned long sig[_NSIG_WORDS];
} sigset_t;
```

Linux 内核提供了 5 个相应的函数用于信号集的操作，对其标准调用格式的说明如下：

```
#include <signal.h>
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signum);
int sigdelset (sigset_t *set, int signum);
int sigismember (const sigset_t *set, int signum);
```

对各个函数的功能、参数和返回值说明如下。

- sigemptyset 函数：用于将 set 参数所指向的信号集设定为空，即不包含任何信号，若调用成功则返回“0”，否则返回“−1”。
- sigfillset 函数：用于将 set 参数所指向的信号集设定为满，即包含所有的信号，若调用成功则返回“0”，否则返回“−1”。
- sigaddset 函数：用于将 signum 参数所代表的信号添加到 set 参数所指向的信号集中，若调用成功则返回“0”，否则返回“−1”。
- sigdelset 函数：用于将 signum 参数所代表的信号从 set 参数所指向的信号集中删除，若调用成功则返回“0”，否则返回“−1”。
- sigismember 函数：用于检查 signum 参数所代表的信号是否位于 set 参数所指向的信号集中，如果是真则返回“1”，如果是假则返回“0”，如果调用出错则返回“−1”。

在信号集进行初始化之后就可在该信号集中增、删特定的信号。对所有以信号集作为参数的函数，都向其传送信号集地址。在后面的学习中将经常使用到信号集。

例如，打算在处理信号 SIGINT 时，只阻塞对 SIGQUIT 信号的处理，可以利用如下的方法：

```
struct sigaction act;
sigemptyset (&act.sa_mask);
sigaddset (&act.sa_mask, SIGQUIT);
sigaction (SIGINT, &act, NULL);
```

8.4 信号的高级操作

在信号的实际应用中常常需要将某个信号精确、定时发送，又或者让某个信号等待一段时间再进行处理，这就涉及了信号的阻塞和挂起操作，也涉及了信号的精确定时操作，在这个过程中还会引入可重入函数的概念。

8.4.1 信号的阻塞和挂起

在 Linux 的信号实际应用中，有时候既不希望进程在接收到信号时立刻中断进程的当前工作，也不希望该信号完全被忽略，而是希望进程延迟一段时间再去调用相关的信号处理函数，可以通过阻塞信号的方法来实现这种需求。

Linux 提供了 sigprocmask 函数和 sigsuspend 函数用于信号的阻塞和挂起。

sigprocmask 函数用于信号的阻塞操作，用于检测或更改进程的信号掩码（signalmask），信号掩码是由被阻塞的发送给当前进程的信号组成的信号集，对其标准调用格式说明如下：

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

sigprocmask 的参数 set 和 oldset 是 sigset_t 类型的指针，用于表示所指向的信号集。set 指向一个信号集时，参数 how 表示 sigprocmask 函数将如何对 set 所指向的信号集以及信号掩码进行操作，其取值及对应的函数功能如表 8.5 所示。当 set 参数为 NULL 时，how 的取值无效。当 oldset 不为 NULL 时，函数 sigprocmask 将进程当前的信号掩码返回给 oldset。

表 8.5 参数 how 的取值及对应功能

how 取值	对应函数功能
SIG_BLOCK	将 set 所指向的信号集中所包含的信号加到当前的信号掩码中，即信号掩码与 set 信号集做逻辑或运算
SIG_UNBLOCK	将 set 所指向的信号集中所包含的信号从当前的信号掩码中删除，即信号掩码与 set 信号集做逻辑减运算
SIG_SETMASK	设定新的当前信号掩码为 set 所指向的信号集中所包含的信号，即以 set 信号集对信号掩码进行赋值操作

除了让一个信号阻塞外，Linux 同样提供了对信号进行挂起操作的函数 sigsuspend，在调用该函数之后，进程停止在该处，等待着开放信号的唤醒。系统在接收到信号后，马上就把现在的信号集还原为原来的，然后调用处理函数。

对 sigsuspend 函数的标准调用格式说明如下：

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

进程的信号屏蔽字设置为由参数 sigmask 指向的值。在捕捉到一个信号或发生了一个会终止该进程的信号之前，该进程也被挂起。如果捕捉到一个信号而且从该信号处理程序返回，则 sigsuspend

返回，并且该进程的信号屏蔽字设置为调用 sigsuspend 之前的值。如果函数调用出错则返回“-1”，同时 errno 被设置为 EINTR。


注　意

此函数没有成功返回值。如果它返回到调用者，则总是返回-1，并且 errno 设置为 EINTR 表示一个被中断的系统调用。

8.4.2　信号的精确定时

在前面的内容中介绍了使用 alarm 函数来对信号进行定时操作，如果希望使用更加精确的定时操作，可以使用 setitimer 函数，对其标准调用格式说明如下：

```
#include <sys/time.h>
int setitimer(int which, const struct itimerval *new_value,struct itimerval *old_value);
```

参数 which 用于指定定时器类型，其支持 3 种类型的定时器，如表 8.6 所示。

表 8.6　setitimer 的 which 参数说明

which 取值	定时器类型	发生信号
ITIMER_REAL	设定绝对时间，即系统的时间	SIGALRM
ITIMER_VIRTUAL	设定程序执行时间，只有在用户模式下才可跟踪时间	SIGVTALRM
ITIMER_PROF	从用户进程开始后开始计时	SIGPROF

参数 new_value 和 old_value 为指向时间参数的结构体指针，itimerval 的结构原型如下：

```
struct itimerval
{
    struct timeval it_interval;   /*计时器重新启动的间歇值*/
    struct timeval it_value;      /*计时器安装后首先启动的初始值*/
};
```

成员 it_interval 和 it_value 也是 timeval 类型的结构体：

```
struct timeval
{
    long tv_sec;            /*时间的秒数部分*/
    long tv_usec;           /*时间的微秒(1/1000000)部分*/
};
```

setitimer 函数将 value 指向的结构体设置为计时器的当前值，如果 old_value 不是 NULL，将返回计时器原有值，其若调用成功则返回 0，若出错则返回-1。

例 8.10 是一个 setitimer 函数的应用实例。

【例 8.10】使用 setitimer 函数进行精确定时

应用代码每隔 1 秒便会调用信号处理函数 signalDeal 打印当前系统的时间和日期，在该函数中，使用了另外两个函数：gettimeofday 和 localtime 函数。

实例的应用代码如下：

```
1   #include <signal.h>
2   #include <time.h>
3   #include <sys/time.h>
4   #include <unistd.h>
5   #include <stdio.h>
6   #include <stdlib.h>
7   //这是对信号的处理函数
8   static void signalDeal(int signo)
9   {
10      struct timeval tp;
11      struct tm *tm;
12      gettimeofday(&tp,NULL);           //获得系统当前时间（秒和微秒）
13      tm=localtime(&tp.tv_sec);          //获得当地目前时间和日期
14      printf(" sec = %ld \t",tp.tv_sec);   //打印从 UNIX 纪元开始到现在的秒数
15      printf(" usec = %ld \n",tp.tv_usec);  //打印微秒
        printf("%d-%d-%d%d:%d:%d\n",tm->tm_year+1900,tm->tm_mon+1,tm->tm_mday,tm->tm_hour,
        tm->tm_min,tm->tm_sec);           /*打印当地目前时间和日期*/
16  }
17  //时间初始化函数
18  static void InitTime(int tv_sec,int tv_usec)
19  {
20      struct itimerval value;           //定义时间参数结构体 value
21      signal(SIGALRM, signalDeal);      //注册信号 SIGALRM  和信号处理函数
22      value.it_value.tv_sec = tv_sec;   //秒
23      value.it_value.tv_usec = tv_usec;  //微秒
24      value.it_interval.tv_sec = tv_sec;
25      value.it_interval.tv_usec = tv_usec;
26      setitimer(ITIMER_REAL, &value, NULL);
27      //setitimer  发送信号，定时类型为 ITIMER_REAL
28  }
29  //主函数
30  int main(int argc,char *argv[])
31  {
32      InitTime(1,0);                    //每隔 1 秒打印一次
33      while(1)
34      {
35      }
36      exit(0);
37  }
```

将文件保存为 exam810settimer.c，在终端进行编译链接，生成可执行文件 exam810settimer。

```
alloy@ubuntu:~/linuxc/chapter8$ gcc exam810settimer.c -o exam810settimer
```

执行 exam810settimer，可以看到对应的时间参数输出：

```
alloy@ubuntu:~/linuxc/chapter8$ ./exam810settimer
 sec = 1393647645          usec = 398239
```

```
    sec = 1393647646        usec = 398234
    sec = 1393647647        usec = 398234
    sec = 1393647648        usec = 398233
    sec = 1393647649        usec = 398233
^C
```

8.4.3　可重入函数

顾名思义，可重入函数就是可以在运行期间再次被调用的函数，由于 Linux 是一个多任务操作系统，在任务执行期间捕捉到信号并对其进行处理时，进程正在执行的指令序列就被信号处理程序临时中断。如果从信号处理程序返回，则继续执行进程断点处的正常指令序列，重新恢复到断点重新执行的过程中，函数所依赖的环境没有发生改变，也就是说这个函数是可重入的，反之就是不可重入的。

在进程中断期间，系统会保存和恢复进程的上下文，然而恢复的上下文仅限于返回地址、处理器、寄存器等之类的少量上下文，而函数内部使用的诸如全局或静态变量、buffer 等并不在保护之列，所以如果这些值在函数被中断期间发生了改变，那么当函数回到断点继续执行时，其结果就不可预料了。例如一个进程正在执行 malloc 分配堆空间，此时程序捕捉到信号发生中断，在执行信号的处理程序中恰好也有一个 malloc，这样就会对进程的环境造成破坏，因为 malloc 通常为它所分配的存储区维护一个链接表，插入执行信号处理函数时，进程可能正在对这张表进行操作，而信号处理函数的调用刚好覆盖了进程的操作，造成错误。

通常来说，满足下面条件之一的大多数是不可重入函数：

- 使用了静态数据结构。
- 调用了 malloc 函数或 free 函数。
- 调用了标准 I/O 函数：标准 I/O 库的很多实现都以不可重入的方式使用全局数据结构。
- 进行了浮点运算：在许多的处理器/编译器中，浮点运算一般都是不可重入的，这是因为浮点运算大多使用协处理器或者软件模拟来实现。

在实际应用中，可重入函数可能存在以下两种状况：

- 信号处理程序 A 的内外都调用了同一个不可重入函数 B；B 在执行期间被信号打断，进入 A（在 A 中调用了 B），运行完成之后返回 B 的被中断点继续执行，这时 B 函数的环境可能改变，其结果就不可预料了。
- 多线程共享进程内部的资源，如果两个线程 A、B 调用同一个不可重入函数 F，A 线程进入 F 后，线程调度切换到 B，B 也执行了 F，那么当再次切换到线程 A 时，其调用 F 的结果是不可预料的。

在信号处理程序中即使调用可重入函数也需要对一些问题进行处理，例如作为一个通用的规则，当在信号处理程序中调用可重入函数时，应当在其前保存 errno 的值，并在其后恢复这个值，这是因为每个线程只有一个 errno 变量，信号处理函数可能会修改其值，若要了解经常被捕捉到的信号是 SIGCHLD，其信号处理程序通常要调用 wait 函数，而各种 wait 函数都能改变 errno 的值。

以下给出了 Linux 中的可重入函数列表：_exit()、 access()、alarm()、cfgetispeed()、cfgetospeed()、

cfsetispeed()、cfsetospeed ()、chdir()、chmod()、chown()、close()、create()、dup()、dup2()、execle()、execve()、fcntl()、fork()、fpathconf ()、fstat()、fsync()、getegid()、 geteuid()、getgid()、getgroups()、getpgrp()、getpid()、getppid()、getuid()、 kill()、link()、lseek()、mkdir()、mkfifo()、 open()、pathconf()、pause()、pipe()、raise()、read()、rename()、rmdir()、setgid ()、setpgid()、setsid()、setuid()、 sigaction()、sigaddset()、sigdelset()、sigemptyset()、sigfillset()、 sigismember()、signal()、sigpending()、sigprocmask()、sigsuspend()、sleep()、 stat()、sysconf()、tcdrain()、tcflow()、tcflush()、tcgetattr()、tcgetpgrp()、tcsendbreak()、tcsetattr()、tcsetpgrp()、time()、times()、 umask()、uname()、unlink()、utime()、wait()、waitpid()、write()。

8.5 本章习题

1. 编写一个程序，使用 signal 函数忽略从终端键入“Ctrl+\”时产生的 SIGQUIT 信号。
2. 编写一个程序，使用 raise 函数向进程自身发送一个 SIGABRT 信号，使进程非正常结束。
3. 编写一个程序，使用 pause 函数将进程挂起，直到有 SIGALRM 信号发生时才从 pause 返回。
4. 编写一个程序，使用信号，读入终端输入的字符，并将其中的小写字母转换成大写字母后输出。
5. 编写一个程序，实现同一个信号处理函数对多个信号的处理。
6. 编写一个程序，为进程打印 SIGINT 和 SIGTERM 信号的掩码。

第 9 章 Linux 的进程同步机制
——管道和 IPC

在第 8 章中介绍了 Linux 使用信号机制控制进程同步的方法，例 8.6 和例 8.7 给出了两个使用信号进行进程同步的实例，从中可以看到信号仅仅传输很少的信息量，此外信号机制还存在占用系统开销太大、必须使用涉及 Linux 内核的系统调用、数量有限等缺点，所以本章将介绍其他几种进程同步机制，涉及以下内容：

- 管道的工作原理及其使用方法。
- 命名管道的工作原理及其使用方法。
- Linux 的 System V IPC 机制。
- 消息队列的工作原理及其使用方法。
- 信号量的工作原理及其使用方法。
- 共享内存的工作原理及其使用方法。

9.1 Linux 的管道

管道（Pipe），也称为匿名管道，是 Linux 下最常见的进程间的通信方式之一，它是在两个进程之间实现一个数据流通的通道。管道是一种很经典的进程之间的通信方式，其具有两个缺点：

- 部分系统下的管道是半双工的，数据只能向一个方向流动（这一特征应该根据相应的 Linux 内核来确认）。
- 管道通常来说只能在具有相同祖先的进程间使用，例如父子进程、兄弟进程等。

9.1.1 管道的基本概念

管道是 Linux/UNIX 系统中比较原始的进程间的通信形式，数据以一种数据流的方式在进程间流动。在系统中其相当于文件系统上的一个文件，用于缓存所要传输的数据。在某些特性上又不同于文件，例如，当数据读出后，管道中就没有数据了，但文件没有这个特性。

管道是 Linux 中最古老的进程通信机制，其应用非常广泛，和信号类似，其也提供了相应的操作符“|”，以供用户在 Shell 中使用。

操作符“|”将其前后两个命令连接到一起，前一个命令的输出成为后一个命令的输入，可以支持使用多个“|”连接多个命令，对其标准调用格式说明如下：

```
命令 A|命令 B|命令 C……|命令 N
```

命令 A 输出既为命令 B 的输入，假如命令 A 为“ls”命令，则这个输出即为当前目录下的文件列表。

在第 8.1.4 小节中介绍了使用“kill - l”命令来查看当前系统中所支持的信号类型列表，如果想在该信号列表中直接查找含有字符串“SIGRTMAX”的信号，可以使用管道操作符“|”来连接“kill - l”和“grep”命令，此时 Shell 创建了“kill-l”和 grep 两个进程，以及这两个进程间的管道，将“kill - l”命令的输出作为“grep”命令的输入，也就是说在信号列表中查找包括“SIGKILL”的信号，其输出如图 9.1 所示。

```
alloy@ubuntu:~/linuxc$ kill -l|grep SIGKILL
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
```

图 9.1　管道命令的应用

9.1.2　管道的实现方法

当一个进程创建一个管道时，Linux 系统内核为使用管道准备了两个文件描述符：一个用于管道的输入，也就是在管道中写入数据；另一个用于管道的输出，也就是从管道中读出数据，然后进程对这两个文件描述符调用正常的系统调用，内核利用这种抽象机制实现了管道这一特殊操作，如图 9.2 所示。

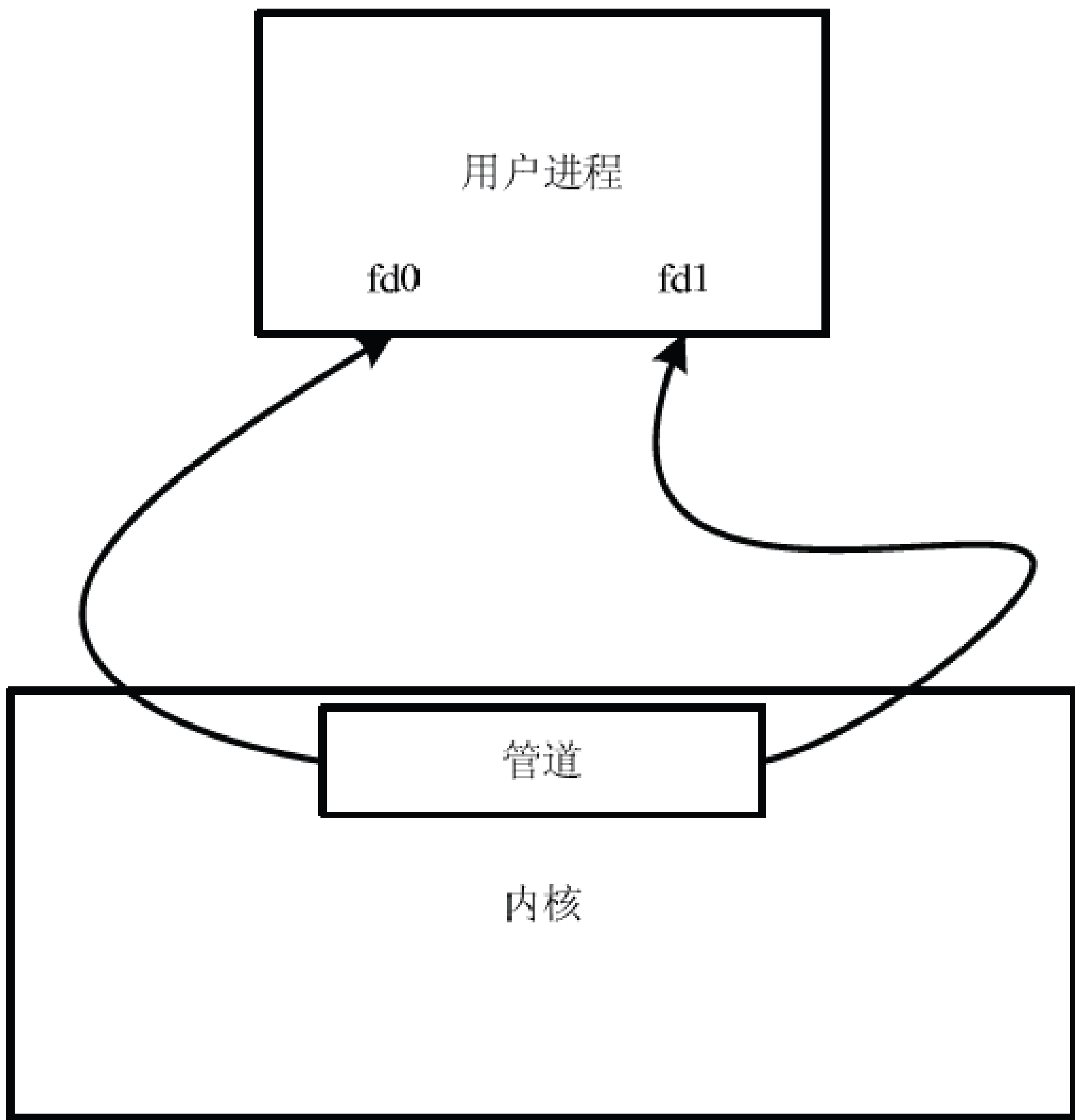


图 9.2　管道的结构

如果一个管道只与一个进程相联系，只实现进程自身内部的通信，则这个管道是毫无意义的。通常情况下，一个创建管道的进程接着就会创建其子进程，由于父子进程可以共享打开文件，子进程将从父进程那里继承到读写管道的文件描述符，这样，父子进程间的通信管道就建立起来了，如图 9.3 所示。

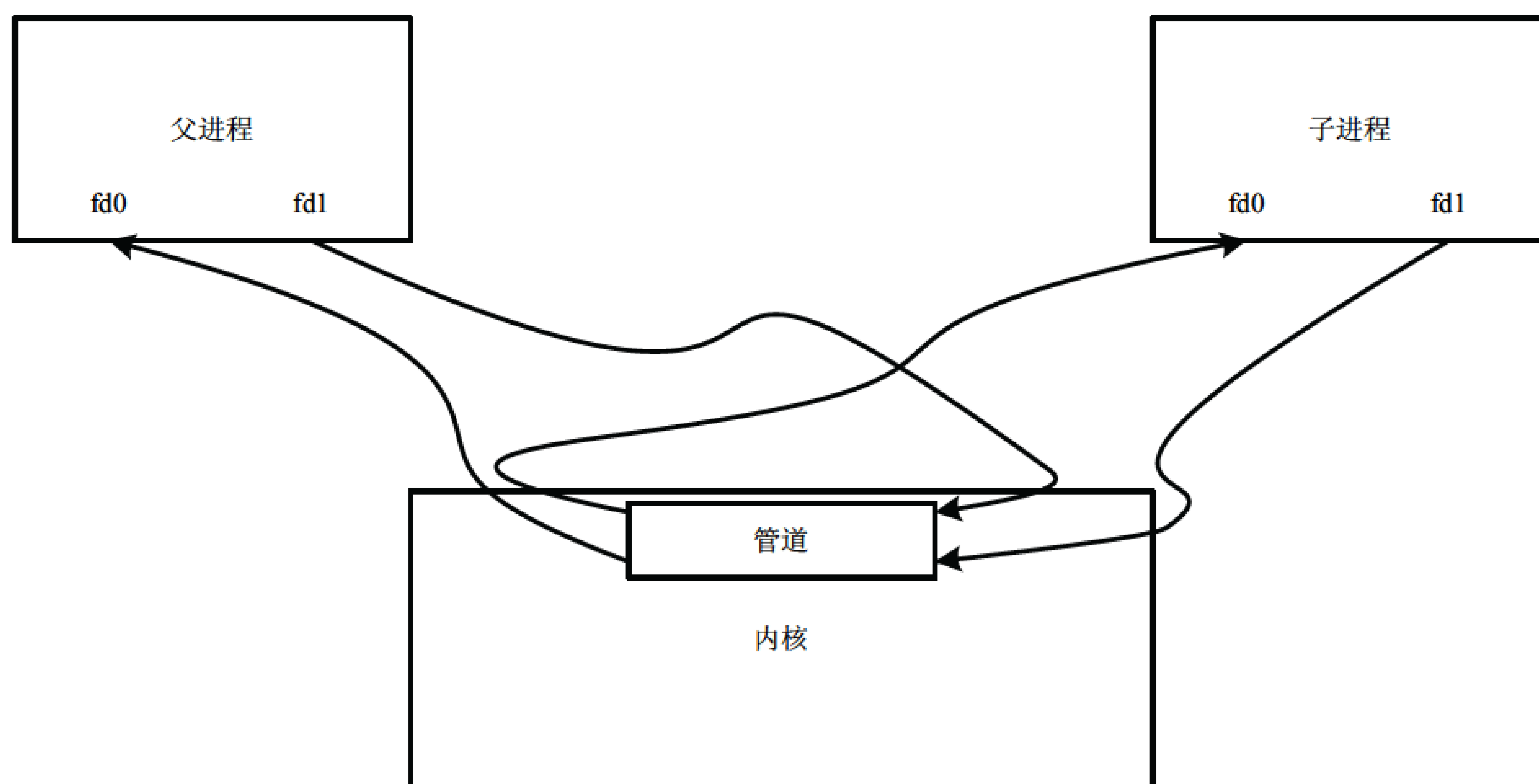


图 9.3　父进程和子进程之间的管道

最后需要确定数据的传输方向，是从子进程传送到父进程，还是从父进程传送到子进程。这一点确定之后，父子进程分别关闭与之无关的描述符。例如数据从子进程传送到父进程，则子进程关闭读管道的描述符，父进程关闭写管道的描述符，这样就建立了从子进程到父进程的通信管道，如图 9.4 所示。

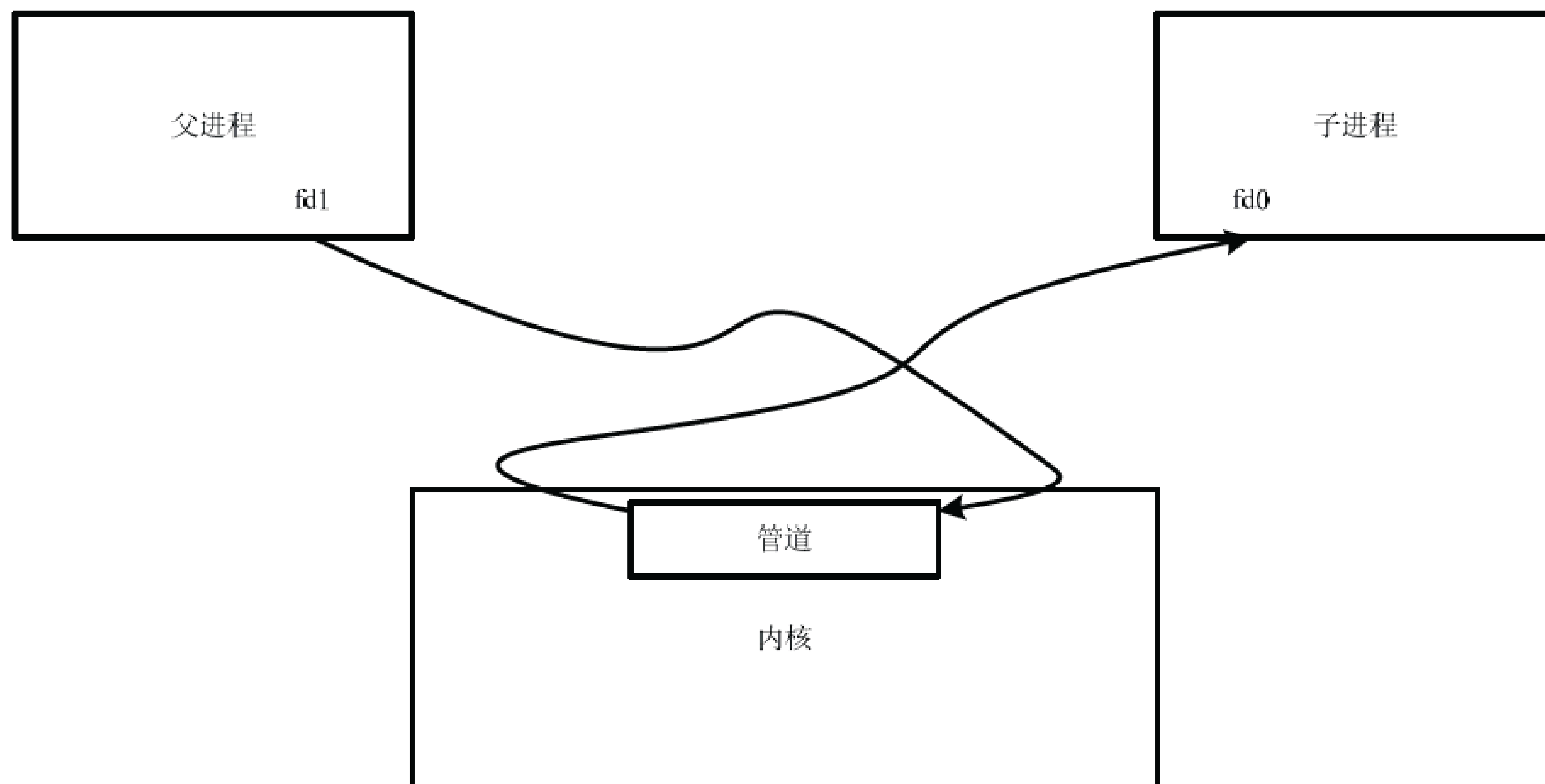


图 9.4　从父进程到子进程的管道

9.1.3　管道的读写操作规则

在建立了一个管道之后即可通过调用相应的文件操作函数（例如 read、write 等）来读写管道，以完成信息的传递。

需要注意的是由于管道的一端已经关闭，在进行相应的操作时需要注意以下三个要点：

- 如果从一个写描述符关闭的管道中读数据，当读完所有的数据后，read 函数返回 0，表明已到达文件末尾。严格来说，只有当没有数据继续写入后，才可以说到达了文件末尾，

所以应该分清到底是暂时没有数据输入，还是已经到达文件末尾，如果是前者，读进程应该等待。若为多进程写、单进程读的情况就更加复杂。

- 如果向一个读描述符关闭的管道中写数据，就会产生 SIGPIPE 信号。不管是忽略这个信号，还是处理它，write 函数都将返回-1。
- 常数 PIPE_BUF 规定了内核中管道缓冲的大小，所以在写管道时要注意这一点。一次向管道中写入 PIPE_BUF 或更少的字符，不会和其他进程写入的内容交错；反之，当存在多个写管道的进程时，向其中写入超过 PIPE_BUF 个字符时，就会产生交错现象。



在 Linux 系统中，可以使用 pathconf 或者 fpathconf 函数来确定 PIPE_BUF 的大小，在 Ubuntu 中这个值是 4096。

9.1.4 管道的特点

Linux 的管道具有以下特点：

- 管道没有名字，所以也称为匿名管道。
- 管道是半双工的，数据只能向一个方向流动；需要双方向通信时，需要建立起两个管道。
- 只能用在父子进程或者兄弟进程之间（具有亲缘关系的进程）。
- 单独构成一种独立的文件系统，管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在于内存中。
- 数据的读出和写入：一个进程向管道中写入的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。
- 管道的缓冲区是有限的（管道只存在于内存中，在管道创建时，为缓冲区分配一个页面大小）。
- 管道所传送的是无格式字节流，这就要求管道的读出方和写入方必须事先约定好数据的格式，例如多少字节算作一个消息（或命令、记录）等。



在实际应用中，由于管道中的数据是无格式的，所以必须采用一个事先设计好的数据格式。

9.2 Linux 的管道操作

Linux 的管道操作包括创建管道和对管道的读写两个部分。

9.2.1 管道的创建

Linux 内核提供了函数 pipe 用于创建一个管道，对其标准调用格式说明如下：

```
#include <unistd.h>
```

```
int pipe(int pipefd[2]);
```

函数的参数 pipefd[2]是一个长度为 2 的文件描述符数组，其中 pipefd[0]是读出端的文件描述符，fd[1]是写入端的文件描述符，也就是说 pipefd[0]只能为读打开，而 pipefd[1]是为写操作打开的。当函数调用成功之后，则自动维护了一个从 fd[1]到 fd[0]的数据通道。

如果函数调用成功，则返回“0”，如果调用失败，则返回“-1”。

例 9.1 是一个使用 pipe 函数创建管道的实例。

【例 9.1】使用 pipe 函数创建管道

应用代码使用 pipe 函数在进程中创建一个管道，其使用了一个 int 类型的数组 fd[2]来作为 pipe 函数的参数，当创建管道成功之后该管道的写入端和读出端的文件描述符分别存放在数组元素 fd[0]和 fd[1]中；主进程使用 printf 函数分别打印这两个文件描述符的值，然后将一个字符串通过管道的写入端写入管道，然后通过管道的读出端读出，再使用 printf 函数输出到屏幕，其流程如图 9.5 所示。

实例的应用代码如下：

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <errno.h>
5   int main(int argc,char *argv[])
6   {
7       int fd[2];                                  //  文件描述符
8       char writebuf[] = "this is a test!\n";      //写缓冲区
9       char readbuf[20];                           //读缓冲区
10      if((pipe(fd)) < 0)                          //创建管道
11      {
12          printf("创建管道失败!\n");
13          exit(0);
14      }
15      write(fd[1],writebuf,sizeof(writebuf) );    //向管道写入端写入数据
16      read(fd[0], readbuf, sizeof(writebuf) );    //从管道读出端读出数据
17      printf ("%s",readbuf );                     //输出字符串
18      printf ("管道的读 fd 是%d,管道的写 fd 是%d \n", fd[0], fd[1]) ;          //打印管道描述符
19      close(fd[0]);                               // 关闭管道的读出端文件描述符
20      close(fd[1]);                               // 关闭管道的写入端文件描述符
21      return 0;
22  }
```

将文件保存为 exam901pipe.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam901pipe。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam901pipe.c -o exam901pipe
```

执行该可执行文件，可以看到首先会输出从管道中读出的字符串，然后在屏幕上分别打印出管道的读、写文件描述符。

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam901pipe
```

```
this is a test!
管道的读 fd 是 3,管道的写 fd 是 4
```

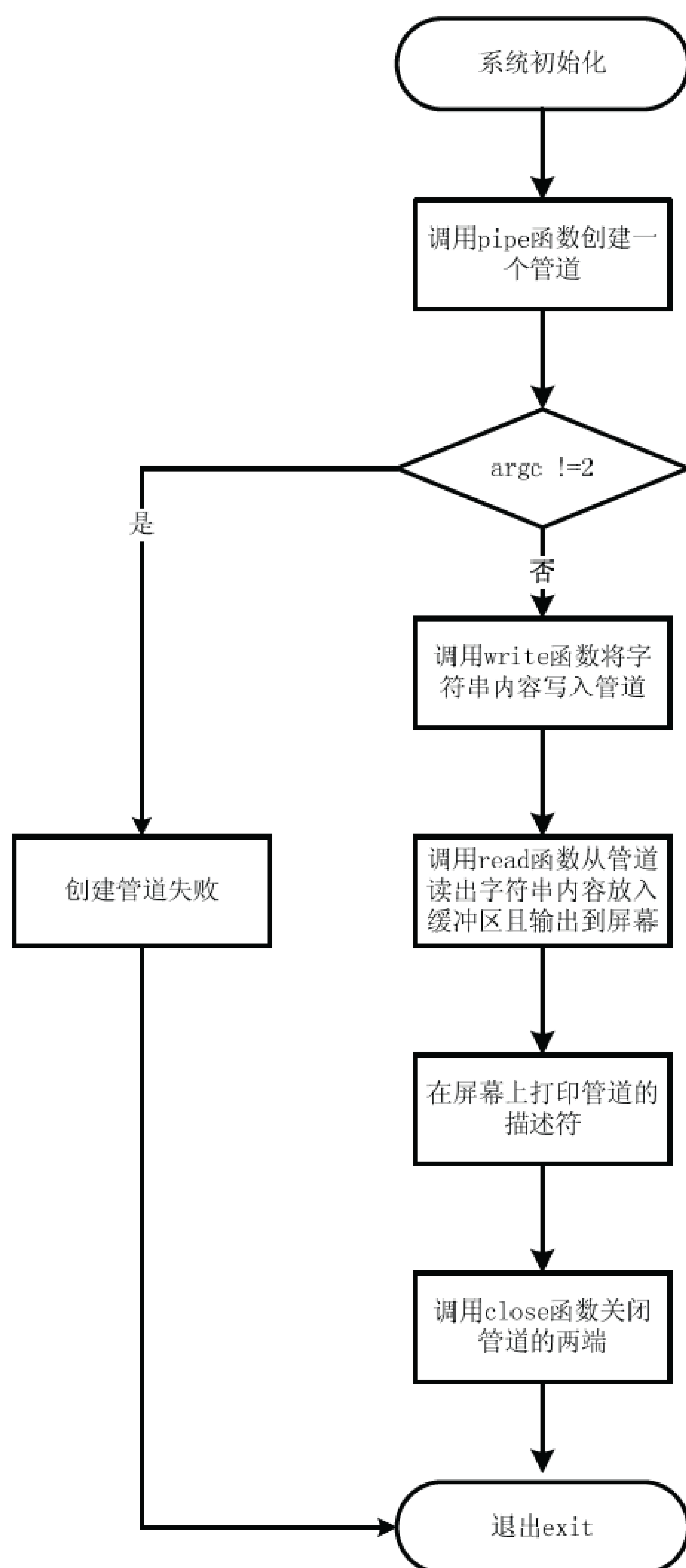


图 9.5　使用 pipe 函数创建管道



注　意

在关闭一个管道的时候必须对管道的两端都执行 close 操作，也就是说要对管道的两个文件描述符都进行相应的操作。

9.2.2　进程的管道通信

在进程中自己创建一个管道是没有意义的，管道的用途是为了在两个不同的进程中进行数据交互，本小节将介绍如何使用管道在不同的进程中进行数据交互，需要注意的是在第 9.1.4 小节中介绍过管道的特点是只能在有亲属关系的进程间使用，即父子进程、兄弟进程等。

1. 在父子进程中使用管道

在父子进程中使用管道的详细步骤如下：

01 在父进程中使用 pipe 函数创建一个管道。

02 在父进程中使用 fork 函数创建一个子进程。

03 在父进程中关闭不使用的管道一端的文件描述符，然后使用对应的写操作函数，例如 write 将对应的数据写入管道。

04 在子进程中关闭不使用的管道一端的文件描述符，然后使用对应的读操作函数，例如 read 将对应的数据从管道中读出。

05 关闭管道的文件描述符。

例 9.2 是一个在父子进程中使用管道的实例。

【例 9.2】在父子进程中使用管道

应用代码定义了一个由 25 个 char 类型变量构成的缓冲区，然后在父进程中创建一个管道，调用 write 函数将一个字符串写入管道，在子进程中调用 read 函数，从管道读出这个字符串并且打印到屏幕上。

实例的应用代码如下：

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <sys/types.h>
4   #include <stdlib.h>
5   #include <errno.h>
6
7   int main(int argc,char *argv[])
8   {
9       int n, fd[2];
10      pid_t pid;
11      char buffer[25];                        //缓冲区
12      if(pipe(fd)<0)                          //创建一个管道，两个文件描述符位于 fd 数组中
13      {
14          printf("创建管道失败!\n ");
15          exit(0);
16      }
17      if((pid=fork())<0)                      //创建一个子进程
18      {
19          printf("创建子进程失败!\n ");
20          exit(0);
21      }
22      else if (pid>0)                         //父进程
23      {
24          close(fd[0]);
25          write(fd[1],"This is a pipe test!\n",22);   //向管道写入数据，注意回车换行符
26      }
```

```
27      else                                        //子进程
28      {
29          close(fd[1]);                           //关闭
30          n = read(fd[0],buffer,25);              //从通道中读出数据
31          printf("%s",buffer);                    //将数据写到标准输出设备
32      }
33      exit(0);
34  }
```

将文件保存为 exam902pipefartherson.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam902pipefartherson。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam902pipefartherson.c -o exam902pipefartherson
```

执行该可执行文件，可以看到对应的字符串输出。

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam902pipefartherson
This is a pipe test!
```

2. 在兄弟进程中使用管道

在兄弟进程中使用管道进行数据交互的方法和在父子进程中类似，只是将对管道进行操作的两个进程更换为兄弟进程即可，在父进程中则关闭该管道。

例 9.3 是一个在兄弟进程中使用管道的应用实例。

【例 9.3】在兄弟进程中使用管道

应用代码在主进程中创建了一个管道和两个子进程，然后在第 1 个子进程中将一个字符串通过管道发送给第 2 个子进程，第 2 个子进程从管道中读出数据，然后将该数据输出到屏幕上。

实例的应用代码如下：

```
 1  #include <unistd.h>
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <sys/types.h>
 5  #include <limits.h>
 6  #include <string.h>
 7  #include <errno.h>
 8  #define BUFSIZE 4096                                        //定义一个最大的读写空间
 9  int main(void)
10  {
11      int fd[2];
12      char buf[BUFSIZE] = "hello!I am your    brother!\n";    // 缓冲区
13      pid_t pid;
14      int len;
15      if ( (pipe(fd)) < 0 )                                   //创建管道
16        {
17          perror("pipe failed\n");
18        }
19      if ( (pid = fork()) < 0 )                               //创建第 1 个子进程
```

```
20          {
21              perror("fork failed\n");
22          }
23          else if ( pid == 0 )                                    //子进程
24          {
25              close ( fd[0] );                                    //关闭不使用的文件描述符
26              write(fd[1], buf, strlen(buf));                     //发送字符串
27              exit(0);
28          }
29          if ( (pid = fork()) < 0 )                               //创建第 2 个子进程
30          {
31              perror("fork failed\n");
32          }
33          else if ( pid > 0 )                                     //父进程
34          {
35              close ( fd[0] );
36              close ( fd[1] );
37              exit ( 0 );
38          }
39          else                                                    //第 2 个子进程中
40          {
41              close ( fd[1] );                                    //关闭管道文件描述符
42              len = read (fd[0], buf, BUFSIZE);                   //读取消息
43              write(STDOUT_FILENO, buf, len);                     //将消息输出到标准输出
44              exit(0);
45          }
46          return 0;
47      }
```

将文件保存为 exam903pipebrother.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam903pipebrother。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam903pipebrother.c -o exam903pipebrother
```

执行该可执行文件，可以看到对应的字符串输出：

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam903pipebrother
hello!I am your   brother!
```

3. 管道的实际应用

在前面两个小节中分别介绍了管道在父子进程和兄弟进程中的应用方法，本小节给出两个管道的实际应用实例。

【例 9.4】管道的实际应用

应用代码在主进程中打开一个由 argv 参数指定的文件，然后调用 pipe 函数创建了一个管道，再调用 fork 函数创建两个子进程。在子进程 1 中每隔 1 秒通过管道向子进程 2 发送一个字符串；在子进程 2 中则从管道读出该字符串，然后写入文件中，其流程如图 9.6 所示。

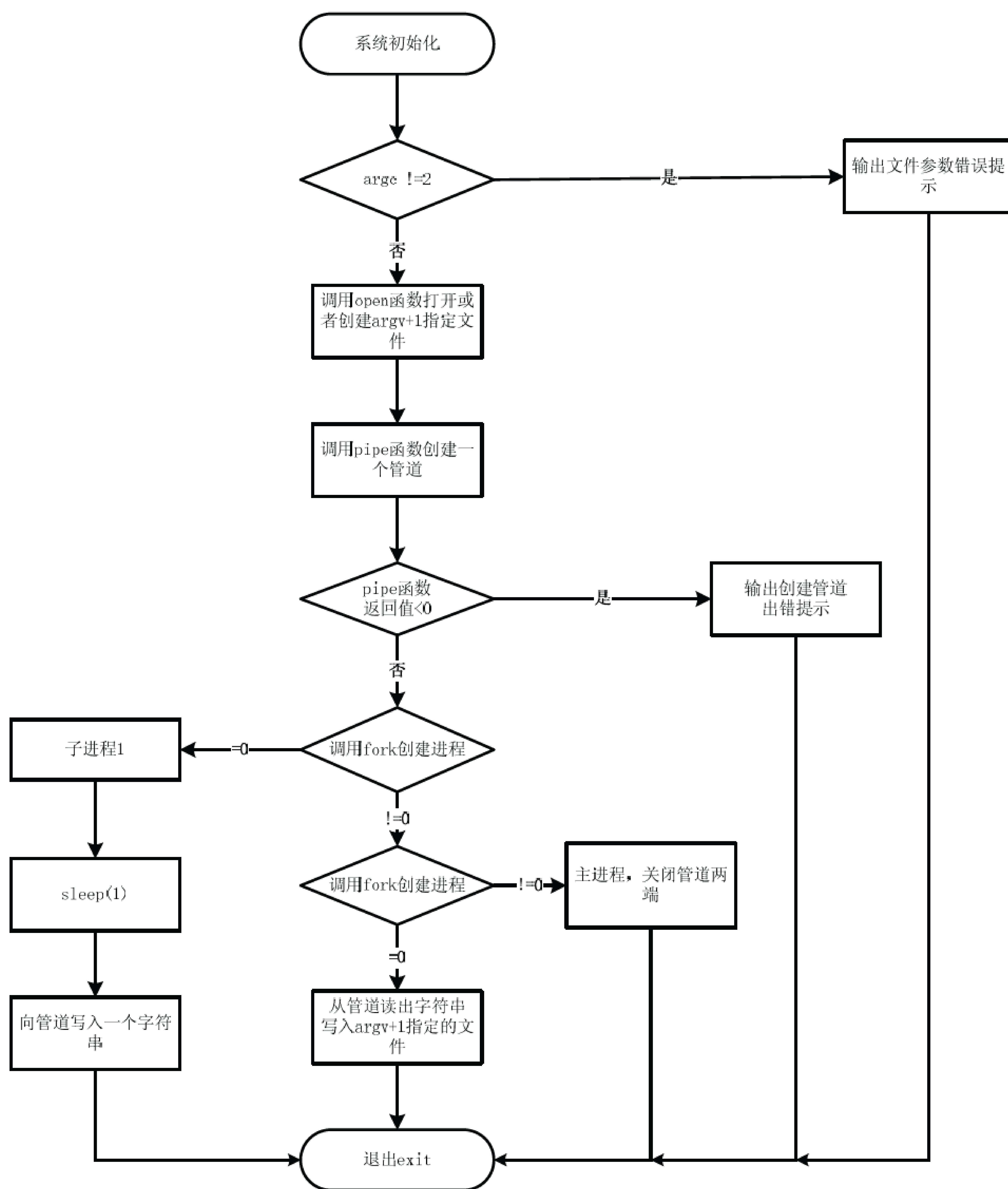


图 9.6　管道的实际应用

实例的应用代码如下：

```
1  /*
2  主进程创建 2 个子进程，子进程 1 每隔 1 秒向子进程 2 发送一个
3  字符串，子进程接收到该字符串之后将其写入一个指定的文件
4  */
5  #include <stdlib.h>
6  #include <unistd.h>
7  #include <stdio.h>
8  #include <signal.h>
9  #include <time.h>
```

```
10  #include <fcntl.h>
11  #include <string.h>
12  #include <sys/types.h>
13
14  //以下为主函数
15  int main(int argc,char *argv[])
16  {
17     pid_t pid1,pid2;                          //进程的 ID
18     int fd;                                   //文件描述符
19     char writebuf[] = "this is a test!\n";    //待写入字符串
20     char readbuf[25];                         //读缓冲区
21     int writecounter = 0;                     //用于记录写入的偏移量
22     int temp = 0,seektemp = 0,j = 0;          //都是用于计算文件偏移的临时变量
23     int pipefd[2];                            //管道的文件描述符
24     if (argc != 2)                            //如果参数不正确
25     {
26        printf("请输入正确的文件参数。\n");
27        return 0;
28     }
29     fd = open(*(argv+1),O_RDWR|O_CREATE,S_IRWXU);     //打开或者创建一个文件
30     if(pipe(pipefd) < 0)                              //如果创建管道失败
31     {
32        printf("创建管道失败。\n");
33        exit(0);                                       //退出
34     }
35     pid1 = fork();                                    //调用 fork 创建一个新的进程
36     if(pid1 != 0)                                     //主进程
37     {
38        pid2 = fork();                                 //创建第 2 个子进程
39        if(pid2 != 0)                                  //主进程
40        {
41           close(pipefd[0]);                           //关闭管道
42           close(pipefd[1]);
43        }
44        else                                           //这是子进程 2 的操作
45        {
46           close(pipefd[1]);
47           while(1)
48           {
49              read(pipefd[0],readbuf,sizeof(writebuf));    //读管道
50              printf("这是子进程 2.\n");                    //屏幕输出提示
51              if(writecounter == 0)                        //第一次写入
52              {
53                 temp = write(fd,readbuf,strlen(readbuf)); //写入数据
54                 seektemp = lseek(fd,0,SEEK_CUR);          //获得当前偏移量
55                 writecounter++;
56              }
57              else
58              {
```

```
59                  j = strlen(readbuf)*writecounter;
60                  seektemp = lseek(fd,j,SEEK_SET);
61                  temp = write(fd,readbuf,strlen(writebuf));
62                  writecounter++;
63              }
64          }
65      }
66  }
67  else                                            //子进程
68  {
69      close(pipefd[0]);
70      while(1)
71      {
72          sleep(1);
73          write(pipefd[1],writebuf,sizeof(writebuf));     //将字符串写入管道
74      }
75  }
76 }
```

将文件保存为 exam904pipeuse.c，在终端进行编译链接，生成可执行文件 exam904pipeuse。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam904pipeuse.c -o exam904pipeuse
```

执行该可执行文件，将字符串写入 pipeusetest.txt 文件中，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam904pipeuse pipeusetest.txt
alloy@ubuntu:~/linuxc/chapter9$  这是子进程 2.
这是子进程 2.
这是子进程 2.
这是子进程 2.
这是子进程 2.
这是子进程 2.
```

使用 cat 命令可以查看 pipeusetest.txt 文件的内容：

```
this is a test!
this is a test!
this is a test!
this is a test!
this is a test!
this is a test!
this is a test!
```


注 意

这个可执行文件无法使用“Ctrl+C”快捷键或者“Ctrl+\”快捷键退出执行，需要利用 kill 命令直接杀掉进程，对其操作步骤说明如下，读者可以自行分析导致这种情况的原因。

首先使用“ps –aux”命令查看当前正在运行的进程，从中找出 exam904pipeuse 中两个进程对应的进程号（PID），如图 9.7 所示，可以看到进程号为 1213 和 1214。

```
alloy@ubuntu:~/linuxc/chapter9$ ps -aux
```

```
whoopsie  1162  0.0  0.1 202436  5200 ?        Ssl  Mar01   0:00 whoopsie
root      1186  0.0  0.0      0     0 ?        S    16:13   0:00 [flush-7:0]
root      1194  0.0  0.0   7272  1588 ?        S    Mar01   0:00 /sbin/dhclient -d -4 -sf /usr/lib/NetworkM
alloy     1213  0.0  0.0   4164    88 ?        S    16:20   0:00 ./exam904pipeuse pipeusetest.txt
alloy     1214  0.0  0.0   4168    88 ?        S    16:20   0:00 ./exam904pipeuse pipeusetest.txt
root      1223  0.0  0.1 147240  6356 ?        Ss   16:22   0:00 sshd: alloy [priv]
```

图 9.7　exam904pipeuse 对应的进程号

调用 kill 命令进程号为 1213 和 1214 的进程执行退出操作：

```
alloy@ubuntu:~/linuxc/chapter9$ kill 1213
alloy@ubuntu:~/linuxc/chapter9$ kill 1214
```

例 9.5 是另外一个管道的实际应用实例，子进程调用 dup 或 dup2 函数，将管道的文件描述符复制到标准输入或输出上，接着子进程调用 exec 函数运行其他程序，那么这个程序的标准输入或标准输出就成为从管道读入或向管道输出了。

【例 9.5】管道的实际应用

应用代码通过调用 Linux 的“more”命令来实现分页输出一个指定文件，其首先创建一个管道，然后调用 fork 函数创建一个子进程并且使子进程，并且使子进程的标准输入成为管道的读端，最后利用 exec 函数调用 more 命令来对指定文件进行操作。

实例的应用代码如下：

```
 1  #include <unistd.h>
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <sys/types.h>
 5  #include <limits.h>
 6  #include <string.h>
 7  #include <sys/wait.h>
 8  #include <error.h>
 9
10  #define DEF_PAGER "/bin/more"                //定义处理函数
11  #define MAXLINE    4096                      //行的最大字符数
12
13  int main(int argc, char *argv[])
14  {
15     int       n;
16     int       fd[2];
17     pid_t     pid;
18     char      *pager, *argv0;
19     char      line[MAXLINE];
20     FILE      *fp;
21     if (argc != 2)   //如果参数不正确
22     {
23        printf("请输入正确的命令:<pathname>\n");
24        exit(1);
25     }
26     if ((fp = fopen(argv[1], "r")) == NULL)       //如果以只读方式打开 argv[1]指向的文件出错
```

```
27      {
28         printf("不能打开文件%s", argv[1]);
29         exit(1);
30      }
31      if (pipe(fd) < 0)   //创建管道失败
32      {
33         printf("创建管道失败\n");
34         exit(0);
35      }
36      if ((pid = fork()) < 0)                          //创建子进程失败
37      {
38         printf("创建子进程失败\n");
39         exit(0);
40      }
41      else if (pid > 0)
42      {                                                //父进程
43         close(fd[0]);                                 //关闭读文件描述符
44         //将 argv[1]通过管道发送
45         while (fgets(line, MAXLINE, fp) != NULL)
46         {
47            n = strlen(line);
48            if (write(fd[1], line, n) != n)
49            {
50               printf("写管道失败\n");
51               exit(1);
52            }
53         }
54         if (ferror(fp))                               //如果文件描述符出错
55         {
56               printf("fgets  失败\n");
57               exit(1);
58         }
59         close(fd[1]);
60         if (waitpid(pid, NULL, 0) < 0)
61         {
62               printf("waitpid 失败\n");
63               exit(1);
64         }
65         exit(0);
66      }
67      else                                             //子进程
68      {
69         close(fd[1]);
70         if (fd[0] != STDIN_FILENO)
71         {
72            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
73            {
74               printf("dup2 到标准输入失败\n");
75               exit(1);
```

```
76              }
77              close(fd[0]);      /* don't need this after dup2 */
78          }
79          //exec 函数的参数
80          if ((pager = getenv("PAGER")) == NULL)
81          {
82              pager = DEF_PAGER;
83          }
84          if ((argv0 = strrchr(pager, '/')) != NULL)
85          {
86              argv0++;
87          }
88          else
89          {
90              argv0 = pager;
91          }
92          if (execl(pager, argv0, (char *)0) < 0)
93          {
94              printf("调用 execl 失败%s", pager);
95              exit(1);
96          }
97      }
98      exit(0);
99  }
```

将文件保存为 exam905pipemore.c，在终端使用 gcc 编译链接，生成可执行文件 exam905pipemore。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam905pipemore.c -o exam905pipemore
```

对在例 9.4 中生成的文本文件 pipeusetest.txt 使用该可执行文件，可以看到打印出对应的输出。

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam905pipemore pipeusetest.txt
this is a test!
this is a test!
this is a test!
this is a test!
this is a test!
this is a test!
```

9.2.3　管道的高级应用

在第 9.2.1 小节的应用实例中可以看到 pipe 函数和 fork 函数通常是配合起来使用的，Linux 内核同样提供了“合二为一”的函数用于对应的操作，对其标准调用格式说明如下：

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

popen 函数和 pclose 函数必须配合使用，类似于 fopen 和 fclose 函数的组合。

函数 popen 用于创建管道，内部调用 fork 和 exec 函数执行命令行 cmdstring，返回一个 FILE 结构的指针，即用于访问管道的指针。

popen 中的参数“const char *cmdstring”就是一个命令行。所有的 shell 命令行参数和选项都可以使用，例如其可以使用如下的命令行调用：

```
popen("ls *.*","r");
popen("sort > /tmp/foo","w");
popen("sotr | uniq | more", "w");
```

popen 中的参数“const char *type”用于指出管道的类型。如果管道是以类型“r”打开的，那么这个管道的输入端将连接到命令行 cmdstring 的标准输出端，此时，命令行的输出可以从管道中读入。反之，如果管道是以类型“w”打开的，那么这个管道的输出端将连接到命令行的标准输入端。此时，向管道中写入的数据就成为命令行的输入数据。可以看到，type 的作用与 fopen 和 fclose 中的相同，可以取“r”或“w”，表示管道可读或可写，但决不可以既可读又可写。在 Linux 系统下，规定管道的打开方式取决于 type 的第一个字符，例如 type 为“rw”，那么管道就以“r”方式打开，即以可读方式打开。

函数 pclose 是用来关闭管道的，它关闭标准输入输出流，等待命令行执行完毕后返回结束时的状态。如果 shell 不能执行这个命令行，结束时的状态就如同在 shell 中执行了 exit 函数。

例 9.6 是一个 popen 和 pclose 函数的应用实例。

【例 9.6】使用 popen 函数创建管道

应用代码调用 popen 函数创建了管道，然后通过该管道将“ls -l”命令的输出发送到了 argv 参数指定的文件中，其使用了第 6 章中介绍的流操作函数 fread 和 fwrite 进行相应的操作。

实例的应用代码如下：

```
 1  #include <sys/types.h>
 2  #include <unistd.h>
 3  #include <stdlib.h>
 4  #include <stdio.h>
 5  #include <string.h>
 6
 7  int main(int argc,char *argv[])
 8  {
 9      FILE    *stream;
10      FILE    *wstream;                    //定义两个文件流
11      char    buf[1024];                   //定义缓冲区
12      if(argc != 2)                        //如果文件参数不正确
13      {
14          printf("请输入正确的文件参数\n");
15          exit(1);
16      }
17      memset(buf,'a',sizeof(buf));         //初始化 buf，以免后面写入乱码到文件中
18      stream = popen("ls -l", "r" );
        //将“ls  —l”命令的输出通过管道读取（“r”参数）到 FILE* stream
19      wstream = fopen(*(argv+1), "w+");    //新建一个指定的文件
```

```
20          fread(buf, sizeof(char), sizeof(buf),    stream);
21            //将刚刚 FILE* stream 的数据流读取到 buf 中
22          fwrite(buf, 1, sizeof(buf), wstream );
//将 buf 中的数据写到 FILE*wstream 对应的流中，也是写到文件中
23          pclose(stream );
24          fclose(wstream );                                  //关闭退出
25          return 0;
26   }
```

将文件保存为 exam906popen.c，在终端调用 gcc 进行编译链接，生成可执行文件 exam906popen。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam906popen.c -o exam906popen
```

运行可执行文件 exam906popen，使用 popentest.txt 文本文件作为其输出文件。

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam906popen popentest.txt
```

使用“cat –n”命令查看刚刚生成的 popentest.txt 文本文件的内容，可以看到最后有一段全部为“a”字符的乱码，这是因为缓冲区 buf[1024]利用字符 a 进行了初始化操作。

```
alloy@ubuntu:~/linuxc/chapter9$ cat popentest.txt -n
  1   总用量 24078
  2   -rwxrwxr-x 1 alloy alloy      8747   3 月    1 22:09 exam901pipe
  3   -rw-rw-r-- 1 alloy alloy      827    3 月   1 22:09 exam901pipe.c
  4   -rwxrwxr-x 1 alloy alloy      8697   3 月    2 13:37 exam902pipefartherson
  5   -rw-rw-r-- 1 alloy alloy      798    3 月   1 22:42 exam902pipefartherson.c
  6   -rwxrwxr-x 1 alloy alloy      8754   3 月    2 13:44 exam903pipebrother
  7   -rw-rw-r-- 1 alloy alloy     1308    3 月   1 23:14 exam903pipebrother.c
  8   -rwxrwxr-x 1 alloy alloy      8900   3 月    2 16:18 exam904pipeuse
  9   -rw-rw-r-- 1 alloy alloy     2332    3 月   2 14:02 exam904pipeuse.c
 10   -rwxrwxr-x 1 alloy alloy     13200   3 月    2 16:41 exam905pipemore
 11   -rw-rw-r-- 1 alloy alloy     2121    3 月   2 14:46 exam905pipemore.c
 12   -rwxrwxr-x 1 alloy alloy      8803   3 月    2 16:59 exam906popen
 13   -rw-rw-r-- 1 alloy alloy      999    3 月   2 14:57 exam906popen.c
 14   -rwx------ 1 alloy alloy      24489600   3 月    2 16:27 pipeusetest.txt
 15   -rw-rw-r-- 1 alloy alloy        0    3 月   2 16:59 popentest.txt
 16   aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaalloy
      @ubuntu:~/linuxc/chapter9$
```

在当前目录下使用“ls–l”命令查看目录情况，和 popentest.txt 文件内容进行比较，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter9$ ls -l
总用量 24078
-rwxrwxr-x 1 alloy alloy      8747   3 月    1 22:09 exam901pipe
-rw-rw-r-- 1 alloy alloy      827    3 月   1 22:09 exam901pipe.c
-rwxrwxr-x 1 alloy alloy      8697   3 月    2 13:37 exam902pipefartherson
-rw-rw-r-- 1 alloy alloy      798    3 月   1 22:42 exam902pipefartherson.c
-rwxrwxr-x 1 alloy alloy      8754   3 月    2 13:44 exam903pipebrother
-rw-rw-r-- 1 alloy alloy     1308    3 月   1 23:14 exam903pipebrother.c
```

```
-rwxrwxr-x 1 alloy alloy       8900   3 月    2 16:18 exam904pipeuse
-rw-rw-r-- 1 alloy alloy       2332   3 月    2 14:02 exam904pipeuse.c
-rwxrwxr-x 1 alloy alloy      13200   3 月    2 16:41 exam905pipemore
-rw-rw-r-- 1 alloy alloy       2121   3 月    2 14:46 exam905pipemore.c
-rwxrwxr-x 1 alloy alloy       8803   3 月    2 16:58 exam906popen
-rw-rw-r-- 1 alloy alloy        999   3 月    2 14:57 exam906popen.c
-rwx------ 1 alloy alloy   24489600   3 月    2 16:27 pipeusetest.txt
```

9.3　Linux 的命名管道

从第 9.1 节中可以知道 Linux 的管道只能在有亲缘关系的进制之间实现通信，所以如果两个“毫无关系”的进程需要进行数据交换时就不能使用管道，但是 Linux 内核提供了另一种“管道”可以实现这种功能，其被称为命名管道（Named Pipe）或者先进先出队列（FIFO）。

9.3.1　命名管道的基本概念

命名管道不同于管道之处在于它提供一个路径名与之关联，以命名管道的文件形式存在于文件系统中。这样，即使与命名管道的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过命名管道相互通信（能够访问该路径的进程以及命名管道的创建进程），因此通过命名管道不相关的进程也能交换数据。



注　意

管道和命名管道都是实实在在的文件，但是前者没有公开的文件名，用户在文件系统中不能直接观察到并且访问到它；命名管道则是以普通文件的形式存在的，任何进程都可以将其当成一个普通文件进行处理。

总之，命名管道区别于管道主要体现在以下两点：

- 命名管道可以用于任何两个进程间的通信，并不限制这两个进程同源，因此命名管道的使用比管道的使用要灵活方便得多。
- 命名管道作为一种特殊的文件存放于文件系统中，而不像管道一样存放于内存（使用完毕后消失）。当进程对命名管道的使用结束后，命名管道依然存在于文件系统中，除非对其进行删除操作，否则该命名管道不会消失。

命名管道的出现，极好地解决了系统在应用过程中产生的大量中间临时文件的问题。命名管道可以被 shell 调用，使数据从一个进程过渡到另一个进程，系统不必为中间通道而清理不必要的垃圾，或者去释放该通道的资源，它可以被留做后来的进程使用。

另外，需要注意的是，命名管道严格遵循先进先出的规则，对管道及命名管道的读总是从开始处返回数据，对它们的写则把数据添加到末尾，所以它们不支持诸如 lseek 函数等文件定位操作。

9.3.2　命名管道的工作方式

命名管道通常有如下两种工作方式：

- 命名管道由 shell 命令使用，以便将数据从一条管道传送到另外一条，此时无需创建一个中间临时文件。
- 命名管道用于客户进程和服务器进程的应用程序中，以在客户进程和服务器进程之间传递数据。

1. 命名管道的数据流复制传送

命名管道可以用于复制串行管道之间的数据流，此时不需要将数据写入到中间磁盘文件，因为命名管道具有名字，其可以用于非线性连接。


注 意

管道没有名字，所以只能用于进程之间的线性连接。

图 9.8 是一个对输入流进行两次处理的操作。

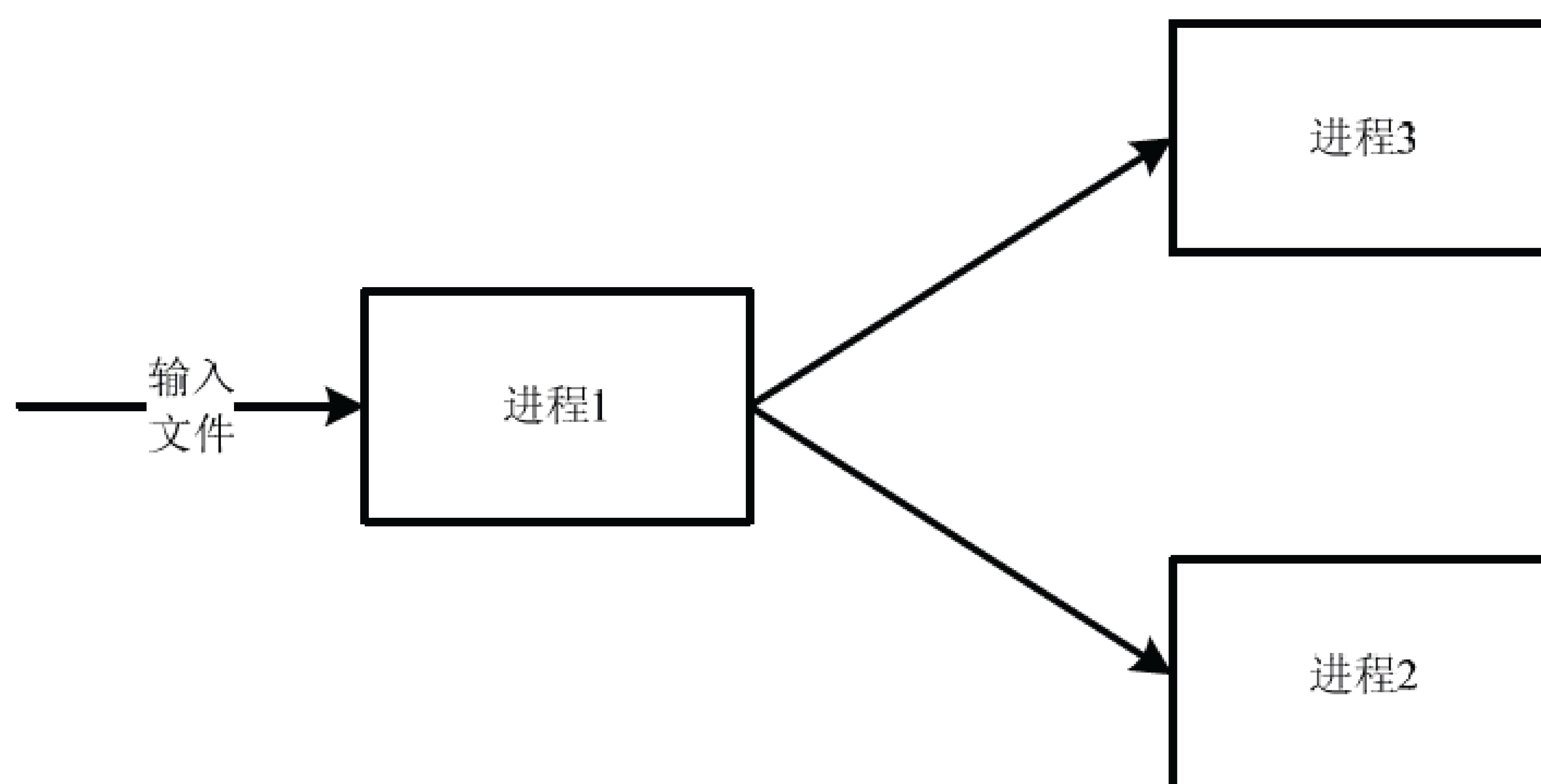


图 9.8 一个对输入流进行两次处理的操作

使用命名管道以及 tee 命令可以实现以上功能，tee 命令从标准输入设备读取数据，将其内容输出到标准输出设备，同时保存成文件，用户可利用 tee 把管道导入的数据存储成文件，甚至一次保存数份文件，对 tee 命令的标准调用格式说明如下。

```
tee [OPTION]... [FILE]...
```

用户可以使用如下的命令序列来实现相应的操作：

```
mkfifo fifo        //创建 fifo 命名管道
prog3 < fifo&    //后台启动进程 3
prog1<infile|tee fifo1|prog3
/*从 fifo 读取数据，然后启动进程 1，并且使用 tee 命令将进程 1 的输出复制到标准输出
和文件 infile*/
```

以上的操作的示意图如图 9.9 所示。

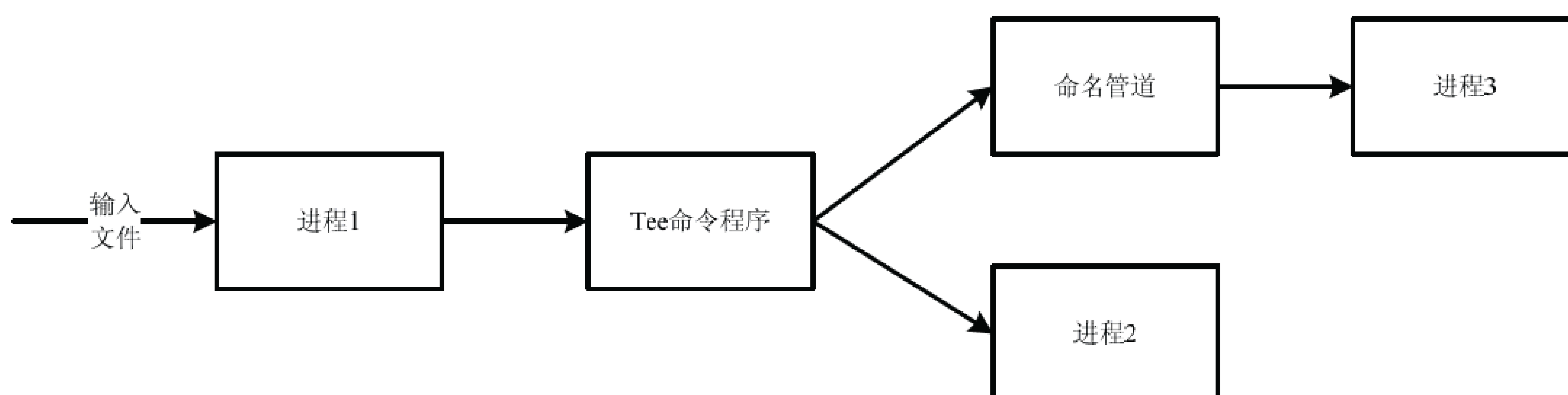


图 9.9　使用命名管道将一个流发送到两个进程

2. 命名管道的终端通信

命名管道还经常用于在客户进程和服务器进程之间传送数据，如果有一个服务器进程需要和多个客户进程相关，则每个客户进程都可以将这个请求写到一个该服务器进程所创建的公共命名管道中，如图 9.10 所示。

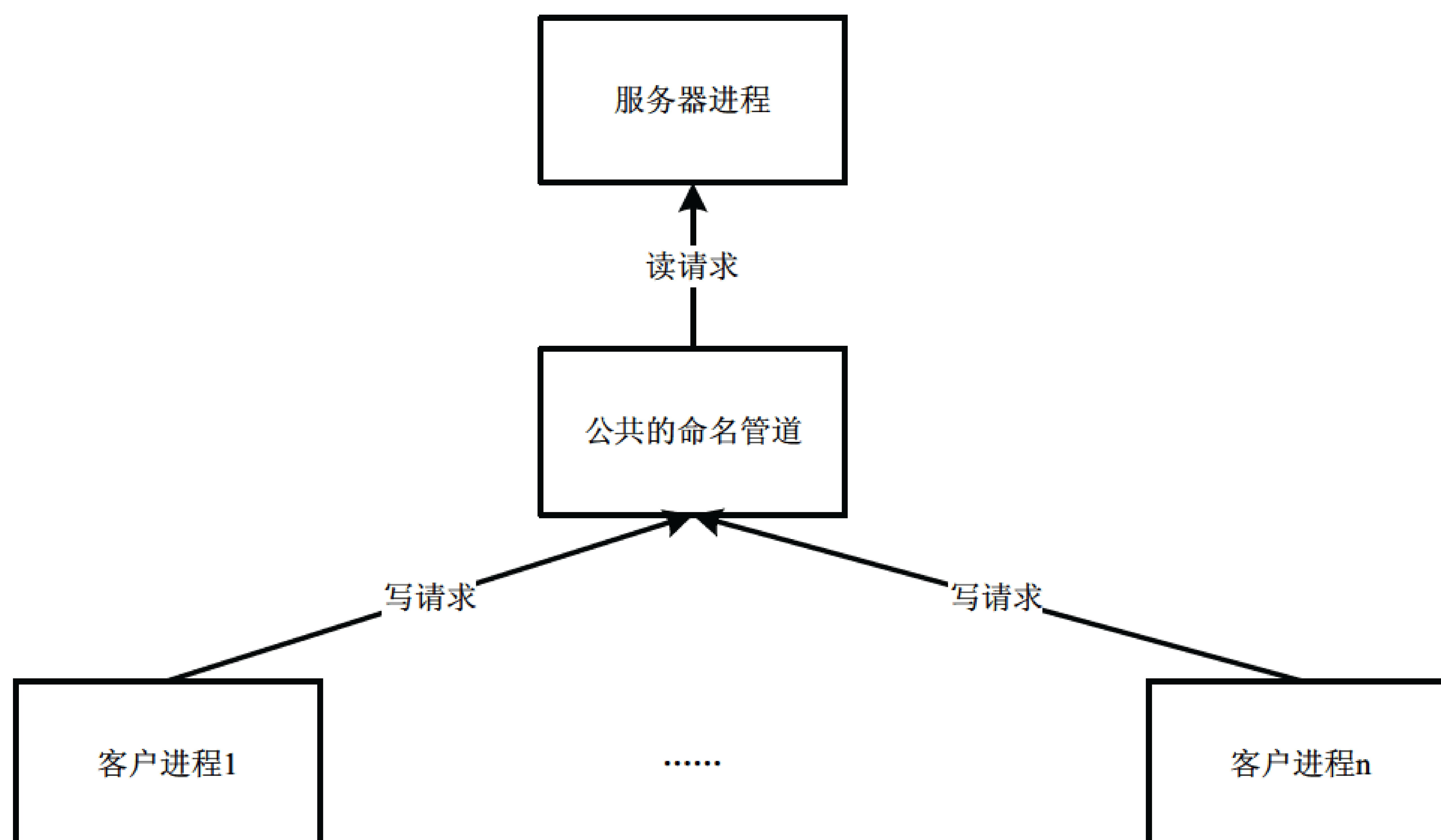


图 9.10　服务器进程和客户进程使用命名管道通信

在这个通信模型中最重要的一个问题是服务器进程如何将应答回馈给各个客户进程，最常见的解决方案是每个客户进程都在其发送的数据包中包含其进程 ID，然后服务器进程根据这些进程 ID 为每个客户进程创建一个命名管道，如图 9.11 所示。

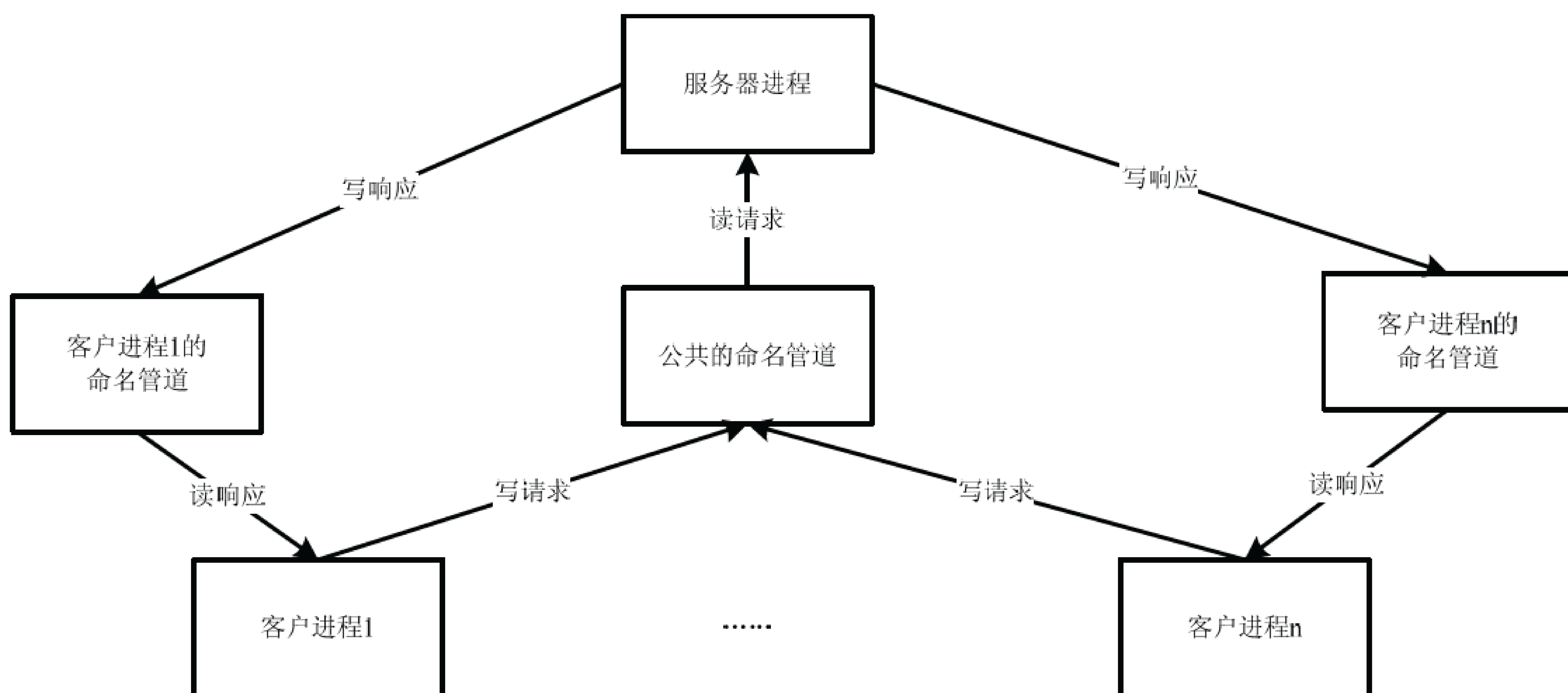


图 9.11 服务器进程和客户进程使用命名管道通信的完整模型

如图 9.11 所示的模型，服务器进程可以只读方式打开公共的命名管道，当客户进程都关闭之后，服务器进程会在这个公共的命名管道中读到一个 EOF（文件结束标志）。这种工作模型具有如下两个缺点：

- 服务器进程不能判断一个客户进程是否会崩溃终止，此时为响应客户进程所建立的客户专属命名管道可能会丧失操作者，从而成为系统垃圾。
- 由于客户进程和服务器进程采用“发送-响应”的工作模式，所以服务器进程必须捕捉 SIGPIPE 信号，否则会出现因响应不及时而导致客户进程挂起的情况。

3. 使用命名管道

在 Shell 环境下，可以很简单地识别出命名管道文件。文件名后面紧跟着一个竖线，就是命名管道文件的标志。而在程序中，由于命名管道文件是一种特殊类型的文件，可以通过 S_ISFIFO 宏来检测。

在 Shell 中可以使用“mkfifo”命令建立一个命名管道，mkfifo 命令的格式如下所示：

```
mkfifo [option] name...
```

其中，在 option 选项中可以选择要创建的命名管道模式，使用形式为“-m mode”，这里的 mode 指出将要创建的命名管道的八进制模式，注意，这里新创建的命名管道会像普通文件一样受到创建进程的 umask 修正。name 表示所要创建的命名管道名称。

关于更详尽的信息，用户可随时使用“man mkfifo”命令查看帮助信息。

一旦建立了一个命名管道，就可以像普通文件那样，对其使用 open、close、read、write、unlink 等文件操作函数，但是由于命名管道是个特殊的文件，不像普通管道那样存在于内核中，仅仅创建并不能立即使用，必须打开才能进行读写操作。读写操作时要特别注意以下几点：

如果没有其他写进程打开一个命名管道，就对其进行读操作时，会产生 SIGPIPE 信号；如果所有的写进程都关闭命名管道，对其的读操作就会认为已到达文件末尾。

在多个写进程的情况下，写交错的现象就有可能发生。与普通管道相同，只要一次写入的字符数不超过 PIPE_BUF，就不会产生写交错现象。

命名管道常常产生阻塞状态，也就是说，如果一个读进程打开命名管道，那么这个进程就要进入阻塞状态，直到其他写进程打开这个管道为止。同样，如果一个写进程打开命名管道，这个进程也会出现阻塞状态，直到其他读进程打开这个管道为止。

如果用户不希望出现这种阻塞状态，可以通过设置 O_NONBLOCK 标志来实现，这样，不管有没有写进程，读打开操作都会立即返回。但是，如果没有读进程，写打开操作就会产生错误。

9.4 Linux 的命名管道操作

9.4.1 创建命名管道

管道没有公开的名字，所以不能进行打开操作，当然其也不需要进行打开操作，但是命名管道是以一个普通文件存在的，用户可以对其进行打开操作（例如调用 open 函数等），但是命名管道的打开与其他文件的打开是有区别的，对其打开规则说明如下：

- 如果当前打开操作是为读而打开命名管道时，若已经有相应进程为写而打开该命名管道，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为写而打开该命名管道（当前打开操作设置了阻塞标志）；或者，成功返回（当前打开操作没有设置阻塞标志）。
- 如果当前打开操作是为写而打开命名管道时，若已经有相应进程为读而打开该命名管道，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为读而打开该命名管道（当前打开操作设置了阻塞标志）；或者，返回 ENXIO 错误（当前打开操作没有设置阻塞标志）。

Linux 内核提供了相应的函数用于创建命名管道，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

mkfifo 函数的 pathname 参数是一个普通的路径名，也就是创建后命名管道文件的名字；mode 参数是文件的操作权限，如果函数调用成功，则返回“0”，否则返回“-1”。

如果 mkfifo 函数的 pathname 参数所指示的文件已经存在，则会返回 EEXIST 错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开命名管道的函数就可以了。通常来说，文件的 I/O 操作函数都可以用于 FIFO，如 close、read、write 等。



注　意

在使用“man”命令查看 mkfifo 函数的相关说明时，必须使用“man 3 mkfifo”。

例 9.7 是一个使用 mkfifo 函数创建命名管道的实例。

【例 9.7】使用 mkfifo 函数创建命名管道

应用代码使用 argv 作为参数调用 mkfifo 函数来创建一个命名管道，通过判断 mkfifo 函数的返回值来判断是否创建成功，如果创建成功则输出对应的提示字符串。

实例的应用代码如下：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5
6  int main(int argc,char *argv[])
7  {
8     mode_t mode = 0755;                    //文件的权限设置
9     if(argc != 2)
10    {
11       printf("请输入正确的文件参数.\n");
12       exit(0);
13    }
14    if(mkfifo(*(argv+1),mode) < 0)         //创建 FIFO 失败
15    {
16       printf("创建 fifo 失败.\n");
17       exit(1);
18    }
19    else
20    {
21       printf("创建 fifo 成功.\n");
22    }
23    return 0;
24 }
```

将文件保存为 exam907mkfifo.c，在终端调用 gcc 进行编译链接，生成 exam907mkfifo 文件。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam907mkfifo.c -o exam907mkfifo
```

执行 exam907mkfifo 文件，创建命名为 myfifotest 的命名管道，可以看到提示创建命名管道成功。

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam907mkfifo myfifotest
创建 fifo 成功.
```

调用“ls–l”命令查看创建成功的 myfifotest 命名管道文件，可以看到对应的输出：

```
alloy@ubuntu:~/linuxc/chapter9$ ls myfifotest -l
prwxr-xr-x 1 alloy alloy 0   3 月   2 22:20 myfifotest
```

9.4.2 读写命名管道

从命名管道中读取数据时必须遵循以下规则：

- 如果一个进程为了从命名管道中读取数据而阻塞打开命名管道，那么称该进程内的读操作为设置了阻塞标志的读操作。
- 如果有进程为写操作打开命名管道，且当前命名管道内没有数据，则对于设置了阻塞标志的读操作来说，将一直阻塞。对于没有设置阻塞标志的读操作来说则返回-1，当前 errno 的值为 EAGAIN，提醒以后再试。
- 对于设置了阻塞标志的读操作说，造成阻塞的原因有两种：当前命名管道内有数据，但有其他进程在读这些数据；另外就是命名管道内没有数据。解阻塞的原因则是命名管道中有新的数据写入，不论新写入数据量的大小，也不论读操作请求多少数据量。
- 读打开的阻塞标志只对本进程的第一个读操作施加作用，如果本进程内有多个读操作序列，则在第一个读操作被唤醒并完成读后，其他将要执行的读操作将不再阻塞，即使在执行读操作时，命名管道中没有数据也一样（此时，读操作返回 0）。
- 如果没有进程为写操作打开命名管道，则设置了阻塞标志的读操作会阻塞。


注 意

如果命名管道中有数据，则设置了阻塞标志的读操作不会因为命名管道中的字节数小于请求读的字节数而阻塞，此时，读操作会返回命名管道中现有的数据量。

向命名管道中写入数据必须符合以下规则：

- 如果一个进程为了向命名管道中写入数据而阻塞打开命名管道，那么称该进程内的写操作为设置了阻塞标志的写操作。
- 对于设置了阻塞标志的写操作，当要写入的数据量不大于 PIPE_BUF 时，Linux 将保证写入的原子性。如果此时管道空闲缓冲区不足以容纳要写入的字节数，则进入睡眠，直到当缓冲区中能够容纳要写入的字节数时，才开始进行一次性写操作。
- 当要写入的数据量大于 PIPE_BUF 时，Linux 将不再保证写入的原子性。命名管道缓冲区一旦有空闲区域，写进程就会试图向管道写入数据，写操作在写完所有请求写的数据后返回。
- 对于没有设置阻塞标志的写操作，当要写入的数据量大于 PIPE_BUF 时，Linux 将不再保证写入的原子性。在写满所有命名管道空闲缓冲区后，写操作返回。
- 当要写入的数据量不大于 PIPE_BUF 时，Linux 将保证写入的原子性。如果当前命名管道空闲缓冲区能够容纳请求写入的字节数，则写完后成功返回；如果当前命名管道空闲缓冲区不能够容纳请求写入的字节数，则返回 EAGAIN 错误，提醒以后再写。


注 意

原子操作（atomic operation）指的是由多步组成的操作。简单来讲，操作的原子性是指某一事务中的所有操作要么全部执行，要么全部不执行，不可能只执行所有步骤的一个子集。

9.4.3 进程使用命名管道通信

本小节将给出一个使用命名管道进行数据交互的实例，涉及例 9.8 和例 9.9 两个实例：一个发

送客户端和一个接收服务器端。

【例 9.8】命名管道通信发送客户端

应用代码使用 MYFIFO 作为命名管道，使用 fopen 函数打开对应的命名管道，然后将 argv 中的字符串写入该命名管道中。

实例的应用代码如下：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define FIFO_FILE "MYFIFO"
5
6  int main(int argc, char *argv[])
7  {
8     FILE *fp;
9     int i;
10    if(argc < 2 )                              //如果参数错误
11    {
12       printf("请使用: %s <pathname>\n",argv[0]);
13       exit(1);
14    }
15    if((fp=fopen(FIFO_FILE,"w"))==NULL)        //打开文件
16    {
17       printf("打开文件失败. \n");
18       exit(1);
19    }
20    for(i=1;i<argc;i++)                        //通过管道发送数据
21    {
22       if(fputs(argv[i],fp)==EOF)
23       {
24           printf("写 fifo 失败. \n");
25           exit(1);
26       }
27       if(fputs(" ",fp)==EOF)
28       {
29          printf("写 fifo 失败. \n");
30          exit(1);
31       }
32    }
33    fclose(fp);
34    return 0;
35 }
```

将文件保存为 exam908fifosendstr.c，并且在终端使用 gcc 编译链接，生成可执行文件 exam908fifosend。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam908fifosendstr.c -o exam908fifosend
```

【例 9.9】命名管道通信接收服务器端

和发送客户端类似，应用代码使用 fopen 函数打开命名为 MYFIFO 的管道文件，然后从其中读出对应的字符串并且显示到屏幕上。

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/stat.h>
4   #include <unistd.h>
5   #include <linux/stat.h>
6   #include <errno.h>
7
8   #define FIFO_FILE "MYFIFO"                    //命名管道名称
9
10  int main(int argc,char * argv)
11  {
12    FILE *fp;
13    char readbuf[80];                           //读缓冲区
14                                                // 创建命名管道文件
15    if((fp=fopen(FIFO_FILE,"r"))==NULL)
16    {
17      umask(0);
18      mknod(FIFO_FILE,S_IFIFO|0666,0);
19    }
20    else
21    {
22      fclose(fp);                               //如果存在则关闭 fp
23    }
24    while(1)
25    {
26     //打开命名管道文件
27      if((fp = fopen(FIFO_FILE,"r"))==NULL)
28      {
29        printf("打开 fifo 失败. \n");
30        exit(1);
31      }
32      // 从命名管道中读数据
33      if(fgets(readbuf,80,fp)!=NULL)
34      {
35        printf("接收到字符串::%s \n", readbuf);
36        fclose(fp);
37      }
38      else
39      {
40        if(ferror(fp))   //如果出错
41        {
42          perror("读文件失败.\n");
43          exit(1);
```

```
44              }
45          }
46      }
47      return 0;
48  }
```

将文件保存为 exam909fifogetstr.c，在终端使用 gcc 进行编译，生成 exam909fifogetstr 可执行文件。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam909fifogetstr.c -o exam909fifogetstr
```

在一个终端中运行 exam909fifogetstr 可执行文件，启动服务器端接收：

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam909fifogetstr
```

打开另外一个终端，运行“exam908fifosend+字符串”，向服务器端发送数据：

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam908fifosend this is a test!
alloy@ubuntu:~/linuxc/chapter9$ ./exam908fifosend the second str!
```

此时可以看到服务器端接收到如下的字符串：

```
接收到字符串::this is a test!
接收到字符串::the second str!
```

9.5　Linux 的 System V IPC 机制

System V IPC 机制是 Linux 从 Unix 继承的进程间通信机制，其由消息队列、信号量以及共享内存三种具体实现方法组成，这三种 IPC 通信方式在编程接口和内部实现上都非常类似。

System V IPC 通信机制的三种具体实现方法具有相同的特点，例如都采用类似的控制函数、都采用类似的 ipc_perm 结构、都具有标志符和关键字。

Linux 内核提供了相应的函数用于实现 System V IPC 通信，消息队列、信号量和共享内存三种具体实现方式分别对应不同的头文件和动作操作函数，如表 9.1 所示。

表 9.1　System V IPC 的通信函数

实现方式	头文件	创建或打开 IPC	IPC 控制函数	IPC 操作函数
消息队列	<sys/msg.h>	msgget 函数	msgctl 函数	msgsnd 函数 msgrcv 函数
信号量	<sys/sem.h>	semget 函数	semctl 函数	semop 函数
共享内存	<sys/shm.h>	shmget 函数	shmctl 函数	shmat 函数 shmdt 函数


注　意

可以看到 System V IPC 的三种实现方法中的相应操作函数名都很类似，但是必须注意其对应的头文件是不同的。

9.5.1 System V IPC 的标识符和关键字

1. 标识符

每一个 System V IPC 的结构（消息队列、信号量和共享存储区段）都对应了一个标识符（id），其是一个非负整数，当一个 IPC 结构被创建的时候，和该结构相关的标志符会自动加 1 并且赋予这个结构唯一的内部标识，当这个非负整数累加到溢出的时候，会自动恢复到 0 重新开始。

每个 System V IPC 的进程通信机制中的结构都需要和唯一的一个标识符相联系，如果进程要访问这个 IPC 结构，则需要在 Linux 操作系统中传递这个唯一的引用标识符。例如，要访问某个共享内存段，唯一需要的就是给这个内存段指定标识符，只有通过这个标识符才可以完成相关的操作。

标识符的唯一局限是在对应的 IPC 结构类别内，为了说明这一点，假设“666”是某个消息队列的标识符，那么肯定不会有第二个消息队列的标识符为“666”，但是某个共享内存或者某个信号集的标识符却有可能是“666”。

2. 关键字

标识符是 IPC 结构的内部名称，其在不同结构分类的内部是唯一的，但是对于整个 System V IPC 机制而言却不是唯一的，上一小节的最后一段给出了这种情况的说明，所以为了使多个合作进程能够使用同一个 IPC 结构，需要给 IPC 结构一个唯一的外部名称，这个外部名被称为关键字（key）。

当调用 msgget 函数、semget 函数或者 shmget 函数创建一个 IPC 结构的时候，必须指定一个关键字，关键字的数据类型是 Linux 内核提供的基本系统数据类型 key_t，这是一个长整型数据，其定义位于头文件<sys/types.h>中，由内核转变为标识符。

在实际应用中，IPC 结构的关键字有如下三种获取方式：

- 父进程或者服务器进程在创建一个新的 IPC 结构时使用关键字 IPC_PRIVATE，将内核返回的标识符存放在某处，以供子进程或者客户进程使用。
- 在一个公用头文件中定义一个服务器进程和客户进程都“知道”的关键字，然后服务器进程使用这个关键字来创建一个新的 IPC 结构。
- 服务器进程和客户进程使用同一个路径和项目 ID（一个 0~255 之间的字符值），然后使用 ftok 函数将路径和项目 ID 转换为一个关键字，然后提供给服务器进程和客户进程使用。

3. ftok 函数

fork 函数用于将一个路径和项目 ID 转换为关键字，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

path 参数必须是一个存在的、可以访问的文件路径名，项目 ID 则只有低 8 位有效，如果调用成功则返回一个 key_t 类型的关键字，如果失败返回(key_t)-1。

对于命名同一个文件的所有路径名，当用同样的 ID 调用 ftok 函数时，该函数返回相同的关键

字；当用不同的 ID 调用 ftok 函数时，返回不同的关键字；如果 ID 的低 8 位为 0，则 ftok 函数返回的是一个随机结果。


注　意

ftok 函数返回的关键字是根据文件的物理索引确定的，因此，如果这个文件在删除后又重新创建，则由 ftok 函数返回的关键字也会改变，尽管路径名仍然一样。

虽然使用 IPC 资源通信的进程可以直接利用诸如 1234 这样的整数作为关键字，但它们之间需要在程序编码上保持一致，并且，这样做还有一个更致命的弱点：其他进程也可能使用这个整数作为另外的 IPC 资源关键字。在这种情况下，则有可能导致混乱，因此，最好利用 ftok 函数来生成 IPC 资源的关键字。

ftok 函数的返回值是一个合成的关键字值，对于同一个文件来说其可以生成多个关键字值，从理论上来说同一个文件可以得到最多 256 个关键字值。

在用户调用 semget、msgget、shmget 函数获得关键字值时，其返回的都是对应 IPC 对象的标识符，图 9.12 展示了这个过程。

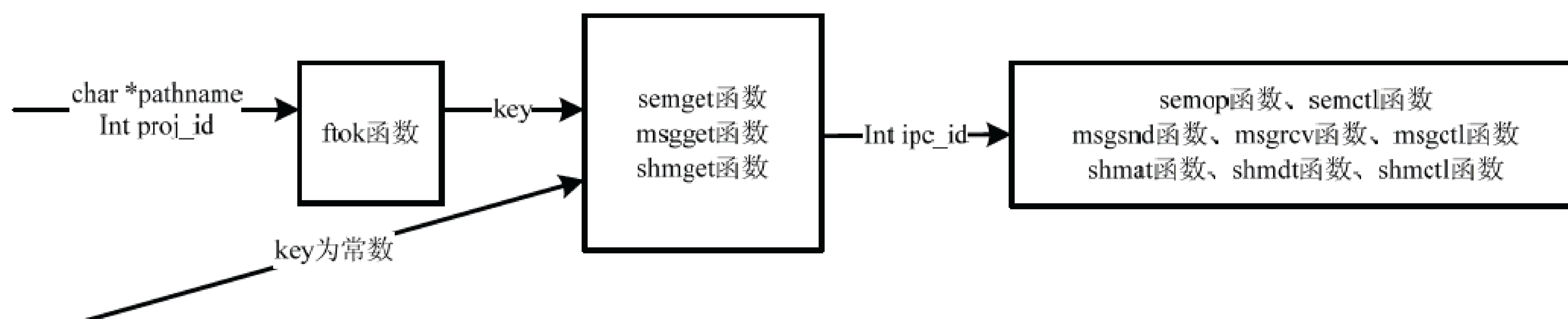


图 9.12　创建关键字

9.5.2　System V IPC 的结构和权限

每一个 IPC 的对象都有一个 ipc_perm 结构与之对应，这个结构中记录了对象的一些信息，如所有者、创建者和权限等。它定义在头文件<sys/ipc.h>中，具体定义如下所示：

```
struct ipc_perm{
uid_t uid;                    /*所有者的有效用户 ID */
        gid_t gid;            /*所有者的有效组 ID */
        uid_t cuid;           /*创建者有效用户 ID */
        gid_t cgid;           /*创建者的有效组 ID */
        mode_t mode;          /*访问权限 */
        ulong seq;            /*应用序号 */
        key_t key;            /*关键字 */
};
```

下面对部分分量的含义进行详细说明。

- uid、gid、cuid 和 cgid：这 4 个分量中记录了 IPC 对象的所有者和创建者的信息，它们是在创建对象时确定下来的，也可以通过系统函数的调用修改它们的值。但是，有权限修改这些值的只能是对象的创建者或超级用户，这与文件系统中的 chown 和 chmod 有

些类似。

- mode：这个分量记录了 IPC 对象的访问权限，它与文件的访问权限有些类似，同样，用户、组用户和其他用户这三类不同用户的权限不同。但是，这些权限中没有可执行权限，并且术语也有些改变。消息队列和共享内存的权限使用术语“可读”、“可写”，而信号量则使用术语“可读”和“可改变”。对 IPC 对象的不同权限说明如表 9.2 所示。

表 9.2　IPC 对象的访问权限

权限	消息队列	信号量	共享内存
用户可读 用户可写	MSG_R MSG_W	SEM_R	SHM_R
组用户可读 组用户可写	MSG_R>>3 MSG_W>>3	SEM_R>>3 SEM_A>>3	SHM_R>>3 SHM_W>>3
其他用户可读 其他用户可写	MSG_R>>6 MSG_W>>6	SEM_R>>6 SEM_A>>6	SHM_R>>6 SHM_W>>6

- seq：这个分量记录了 IPC 对象的应用序号，它并不是确定的值。每次对象被使用，这个值都会增加，直到整数的最大值，然后重新从 0 开始。
- key：这个分量记录了进程通信对象的关键字的值。

msgget、semget、shmget 函数最右边的形参 flag（msgget 中为 msgflg；semget 中为 semflg；shmget 中为 shmflg）为 IPC 对象创建权限，三种函数中 flag 参数的作用基本相同。

IPC 对象创建权限（即 flag）的格式为 0xxxxx，其中 0 表示 8 位制，低三位为用户、属组的读、写、执行权限（执行位不使用），其含义与 ipc_perm 的 mode 相同。

IPC 对象创建权限格式的低三位通常被称为“IPC 对象存取权限”，如“0600”代表只有此用户下的进程才有可读可写权限。IPC 对象存取权限常与下面的 IPC_CREATE、IPC_EXCL 两种标志进行或（|）运算，以完成对 IPC 对象创建的管理，可以把 IPC_CREATE、IPC_EXCL 两种标志称为 IPC 创建模式标志。

下面是两种创建模式标志位于<sys/ipc.h>头文件中的宏定义。

```
#define IPC_CREATE      01000      /* Create key if key does not exist. */
#define IPC_EXCL        02000      /* Fail if key exists. */
```

综上所述，flag 标志由两部分组成：一为 IPC 对象存取权限（含义同 ipc_perm 中的 mode）；二为 IPC 对象创建模式标志（IPC_CREATE、IPC_EXCL），两者进行“|”运算合成 IPC 对象创建权限。

Linux 内核提供了相应的函数来创建一个新的或者访问一个已经存在的 IPC 对象，对其创建或者访问的规则说明如下：

- 指定 key 为 IPC_PRIVATE，操作系统保证创建一个唯一的 IPC 对象。
- 设置 flag 参数的 IPC_CREATE 位，但不设置它的 IPC_EXCL 位时，如果所指定 key 键的 IPC 对象不存在，那就是创建一个新的对象；否则返回该对象。

● 同时设置 flag 的 IPC_CREATE 和 IPC_EXCL 位时，如果所指定 key 键的 IPC 对象不存在，那就创建一个新的对象；否则返回一个 EEXIST 错误，因为该对象已存在。

表 9.3 是对创建 System V IPC 对象的总结。

表 9.3　System V IPC 对象创建总结

flag 创建模式标志	不存在	已经存在
无特殊标志	出错，errno = ENOENT	成功，引用已存在对象
IPC_CREATE	成功，创建新对象	成功，引用已存在对象
IPC_CREATE|IPC_EXCL	成功，创建新对象	出错，errno = EEXIST

在使用 semget、msgget、shmget 创建一个 IPC 对象时，需要指定 flag 标志，在 key 不等于 IPC_PRIVATE 的情况下，flag 标志决定了创建方式和创建后 IPC 对象的存取权限。在 key 等于 IPC_PRIVATE 情况下，flag 标志决定了创建后 IPC 对象的存取权限。如果只是引用一个已经存在的 IPC 对象，只需把 flag 标志设为 0 即可。

图 9.13 是使用相应函数创建或者打开一个 IPC 对象的流程示意图。

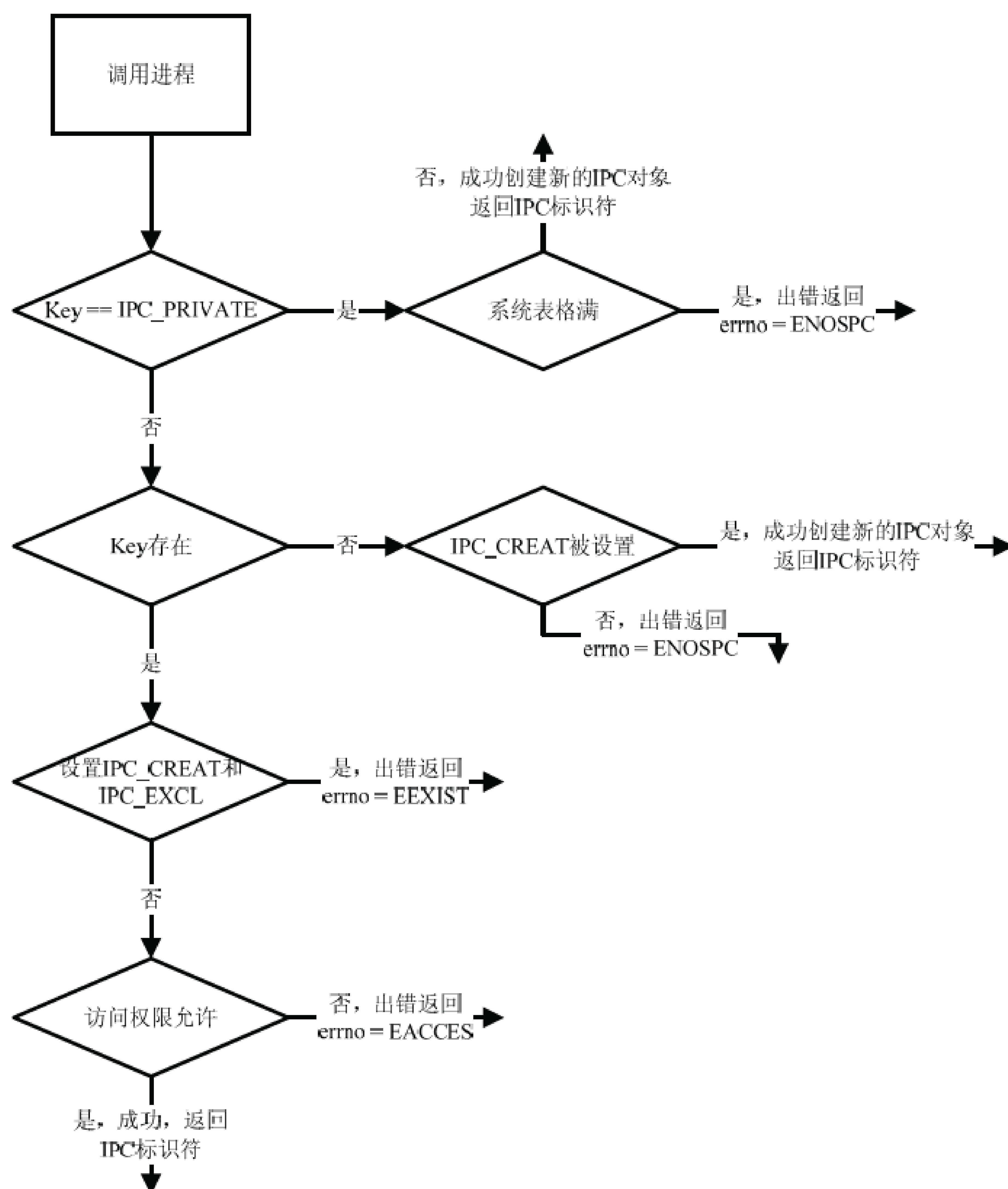


图 9.13　创建一个 IPC 对象流程

9.5.3 System V IPC 的特点

System V IPC 进程通信机制具有以下几个特点：

- IPC 结构是在 Linux 系统范围内起作用的，其没有访问计数机制，其也不会自我删除，停止使用的 IPC 结构会一直保留在系统中，直到被主动删除。
- IPC 结构不能在文件系统中公开访问，主要是因为 IPC 结构没有对应的名字，所以也不能使用文件操作的函数来对这些结构进行操作。
- IPC 结构没有文件描述符，所以不能对其使用多路转接 I/O 函数 select 和 poll，也不能在文件或者设备 I/O 中使用 IPC 结构，还不能一次性使用多个 IPC 结构。

可以使用 ipcs 命令得到当前 Linux 操作系统中 IPC 中的所有对象状态，对该命令的标准调用格式说明如下：

```
ipcs [-asmq] [-tclup]
ipcs [-smq] -i id
ipcs -h
```

以下参数用于指定 IPC 对象类型。

- -m：指定共享内存。
- -q：指定消息队列。
- -s：指定信号量。
- -a：所有的 IPC 结构。

以下用于指定输出格式。

- -t：按时间。
- -p：按照标识符。
- -c：按照创建者。
- -l：按照权限。
- -u：按照概要。

在当前 Linux 中使用“ipcs –a”命令可以看到如下输出：

```
alloy@ubuntu:~/linuxc/chapter9$ ipcs -a

------------ 共享内存段 --------------
键          shmid        拥有者   权限       字节       nattch       状态

--------- 信号量数组 -----------
键          semid        拥有者   权限       nsems

--------- 消息队列 -----------
键          msqid        拥有者   权限       已用字节数  消息
```

9.6　Linux 的消息队列

消息队列是一种以链表式结构组织的数据，存放在内核中，是由各进程通过消息队列标识符来引用的一种数据传送方式。像其他两种 IPC 对象一样，也是由内核来维护。消息队列是三个 IPC 对象类型中最具数据操作性的数据传送方式，在消息队列中可以随意根据特定的数据类型值来检索消息。

9.6.1　消息队列基础

消息队列就是一个消息的链表，每个消息队列都有一个队列头，利用结构 struct msg_queue 来描述。队列头中包含了该消息队列的大量信息，包括消息队列键值、用户 ID、组 ID、消息队列中消息数目等，甚至记录了最近对消息队列读写进程的 ID。用户可以访问这些信息，也可以设置其中的某些信息。

结构 msg_queue 用来描述消息队列头，存在于系统空间中，定义如下：

```
struct msg_queue
{
    struct ipc_perm q_perm;
    time_t q_stime;                      //上一条消息的发送时间
    time_t q_rtime;                      //上一条消息的接收时间
    time_t q_ctime;                      //上一次修改时间
    unsigned long q_cbytes;              //当前队列中的字节数据
    unsigned long q_qnum;                //队列中的消息数
    unsigned long q_qbytes;              //队列的最大字节数
    pid_t q_lspid;                       //上一条发送消息的 pid
    pid_t q_lrpid;                       //上一条接收消息的 pid
    struct list_head q_messages;
    struct list_head q_receivers;
    struct list_head q_senders;
};
```

结构 msqid_ds 用来设置或返回消息队列的信息，存在于用户空间中，定义如下：

```
struct msqid_ds
{
    struct ipc_perm msg_perm;
    struct msg *msg_first;               //队列中第一条消息
    struct msg *msg_last;                //队列中最后一条消息
    time_t msg_stime;                    //上一条消息的发送时间
    time_t msg_rtime;                    //上一条消息的接收时间
    time_t msg_ctime;                    //上一次修改时间
    unsigned long   msg_lcbytes;
    unsigned long   msg_lqbytes;
    unsigned short msg_cbytes;           //当前队列中的字节数据
    unsigned short msg_qnum;             //队列中的消息数
    unsigned short msg_qbytes;           //队列的最大字节数
    pid_t msg_lspid;                     //上一条发送消息的 pid
```

```
    pid_t msg_lrpid;                    //上一条接收消息的 pid
};
```

在 Linux 中消息队列的一些相关配置参数说明如表 9.4 所示。

表 9.4　消息队列的配置参数说明

配置参数	说明
MSGMAX	可以发送消息的最大字节数
MSGMNB	一个队列中最大的字节总数
MSGMNI	系统中允许存在的最大消息队列个数
MSGTQL	系统中允许存在的最大消息个数

图 9.14 是 Linux 操作系统中使用消息队列的示意图，其中 struct ipc_ids msg_ids 是内核中记录消息队列的全局数据结构；struct msg_queue 是每个消息队列的队列头。

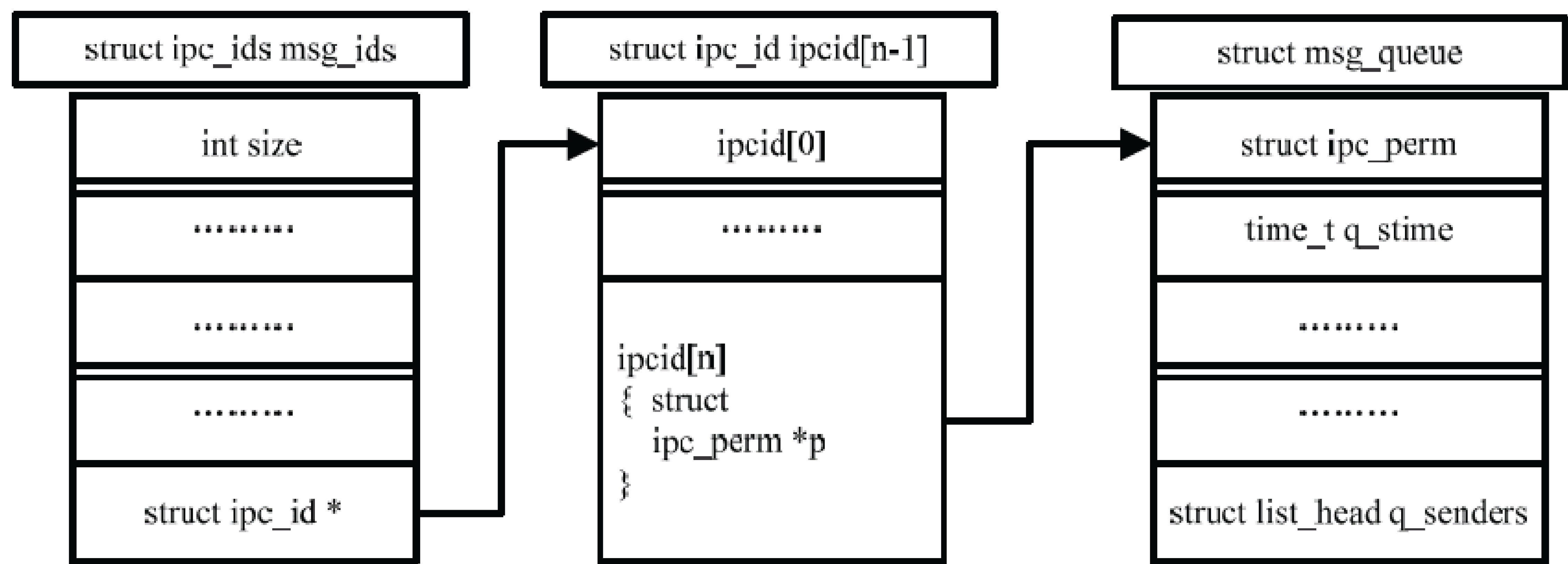


图 9.14　内核数据结构与消息队列

从图 9.14 中可以看到，全局数据结构 struct ipc_ids msg_ids 可以访问到每个消息队列头的第一个成员：struct ipc_perm；而每个 struct ipc_perm 能够与具体的消息队列对应起来是因为在该结构中有一个 key_t 类型成员 key，而 key 可唯一确定一个消息队列。ipc_perm 结构的定义如下：

```
struct ipc_perm
{
    //内核中用于记录消息队列的全局数据结构 msg_ids 能够访问到该结构
    key_t     key;              //该键值唯一对应一个消息队列
    uid_t     uid;              //所有者的有效用户 ID
    gid_t     gid;              //所有者的有效组 ID
    uid_t     cuid;             //创建者的有效用户 ID
    gid_t     cgid;             //创建者的有效组 ID
    mode_t    mode;             //此对象的访问权限
    unsigned long seq;                //对象的序号
};
```

在第 9.6 节中提到过，消息队列是随内核持续的，只有在内核重起或者显示删除一个消息队列时，该消息队列才会真正被删除，因此系统中记录消息队列的数据结构（struct ipc_ids msg_ids）位于内核中，系统中的所有消息队列都可以在结构 msg_ids 中找到访问入口。


注 意

随进程持续的定义为，IPC 一直存在，直至打开 IPC 对象的最后一个进程关闭该对象为止，如管道和有名管道；随着内核的持续定义，IPC 一直持续到内核重新自举或者显示删除该对象为止，如消息队列、信号量以及共享内存等。

9.6.2　消息队列的操作

消息队列的操作包括消息队列的建立、消息队列的控制、消息队列的发送和接收等。

1. 消息队列的建立

Linux 内核提供了 msgget 函数用于创建或者打开一个消息队列，以对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

对函数 msgget 的参数说明如下：

- 当参数 key 的取值为 IPC_PRIVATE 时，不管 flag 为何值，这个函数将创建一个新的消息队列。
- 当参数 key 的取值不为 IPC_PRIVATE 时，操作类型就取决于 flag 的值。如果 flag 中设置了 IPC_CREATE 位，而没有设置 IPC_EXCL 位，则既可能执行打开操作，也可能执行创建操作；当 key 的取值与内核中某个存在的消息队列的关键字相同时，执行打开这个消息队列的操作，返回引用标识符：反之，当 key 的取值不与存在的任何一个消息队列的关键字相同时，就会执行创建消息队列的操作，返回引用标识符。
- 当参数 key 的取值不是 IPC_PRIVATE 时，如果 flag 中同时设置了 IPC_CREATE 和 IPC_EXCL，则只会执行创建消息队列操作：当 key 的取值不与存在的任何一个消息队列的关键字相同时，就会执行创建消息队列的操作，返回引用标识符；当 key 的取值与内核中某个存在的消息队列的访问键相同时，这个函数就会出错返回。

所以，打开存在的消息队列的方法只有一种：将 key 取为要打开的消息队列关键字的值，而 flag 中绝对不能设置 IPC_EXCL。另外，也可以通过 flag 参数设置消息队列的访问权限。当函数调用成功时会返回消息队列的引用标识符，否则返回-1，对其对应的可能 error 列表说明如下。

- EACCES：指定的消息队列已存在，但调用进程没有权限访问它。
- EEXIST: key 指定的消息队列已存在，而 msgflg 中同时指定 IPC_CREATE 和 IPC_EXCL 标志。
- ENOENT：key 指定的消息队列不存在，同时 msgflg 中没有指定 IPC_CREATE 标志。
- ENOMEM：需要建立消息队列，但内存不足。
- ENOSPC：需要建立消息队列，但已达到系统的限制。

当一个新的消息队列被创建时，与之对应的 msqid_ds 结构会按照如下规则进行初始化：

- ipc_perm 结构会被初始化，其中，mode 域的设置会按照 flag 的要求进行。
- msg_qunm、msg_lspid、msg_lrpid、msg_stime 和 msg_rtime 都会被置为 0。
- msg_ctime 被置为当前时间。
- msg_qbytes 被置为系统限制值。

例 9.10 是一个使用 msgget 函数来创建消息队列的实例。

【例 9.10】使用 msgget 创建消息队列

应用代码使用 argv 参数作为 ftok 函数的参数来创建队列的键值，然后使用 msgget 函数来创建或者打开一个队列，通过对 msgget 函数的返回值判断创建队列是否成功。

实例的应用代码如下：

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <sys/types.h>
 4  #include <sys/ipc.h>
 5  #include <sys/msg.h>
 6
 7  int main(int argc,char *argv[])
 8  {
 9      int qid;                                    //队列标志符
10      key_t key;                                  //消息队列键值
11      if(argc < 2)
12      {
13          printf("参数错误.\n");
14          exit(0);
15      }
16      key = ftok(*(argv+1),'a');                  //调用 ftok 函数生成队列键值
17      if(key < 0)
18      {
19          printf("获取队列键值失败.\n");
20          exit(0);
21      }
22      qid = msgget(key,IPC_CREATE | 0666);        //打开或者创建队列
23      if(qid < 0)
24      {
25          printf("创建消息队列出错.\n");
26          exit(0);
27      }
28      else
29      {
30          printf("创建消息队列成功.\n");
31      }
32      return 0;
33  }
```

将文件保存为 exam910msgget.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam910msgget。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam910msgget.c -o exam910msgget
```

使用例 9.8 和例 9.9 中使用的 MYFIFO 文件作为键值参数来创建消息队列。

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam910msgget MYFIFO
创建消息队列成功.
```

使用 ipcs -q 命令来查看当前的消息队列，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter9$ ipcs -q
--------- 消息队列 -----------
键          msqid        拥有者   权限       已用字节数  消息
0x61001829 0             alloy        666           0                 0
```

2. 消息队列的控制

除了对消息队列进行读写操作之外，Linux 内核同样提供了对消息队列进行相应控制的函数 msgctl，其可以用于以下操作：

- 查看消息队列相连的数据结构。
- 改变消息队列的许可权限。
- 改变消息队列的拥有者。
- 改变消息队列的字节大小。
- 删除一个消息队列。

对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

对 msgctl 函数的参数说明如下。

- 参数 msqid：一个正整数，它必须是由 msgget 返回的消息队列 id。
- 参数 cmd：指定要求的操作，如表 9.5 所示。

表 9.5　参数 cmd 说明

值	操作
IPC_STAT	将 msqid 指定的消息队列的内核控制数据结构 msqid_ds 复制至 buf 所指定的用户区域中
IPC_SET	设置 msqid 指定的消息队列的有效用户与组 ID、操作权限以及消息队列的字节数，即设置 msqid 相连的数据结构各成员 msg_perm.uid、msg_perm.gid、msg_perm.mode 和 msg_qbytes 的值为 buf 所指结构中给出的值
IPC_RMID	删除 msqid，以及它所指定的消息队列、相连的数据结构


注　意

执行 IPC_STAT 命令的进程必须具有消息队列的读权限，执行 IPC_SET 和 IPC_RMID 命令的进程只能是消息队列的创建者、拥有者或特权进程。换言之，执行这两个命令的非特权进程必须是有效用户 ID 等于相连数据结构的成员 msg_perm.cuid 或 msg_perm.uid 的进程。此外，只有特权进程才可以增大消息队列的字节数。

- buf 参数：其是一个指向类型为 msqid_ds 的结构，该结构由用户分配存储空间，它用于存放 IPC_STAT 命令的返回结果，或 IPC_SET 命令要设置的值。

如果 msgctl 函数调用成功，则返回“0”，否则返回“-1”，对其对应的错误列表说明如下。

- EACCESS：参数 cmd 为 IPC_STAT，无权限读取该消息队列。
- EFAULT：参数 buf 指向无效的内存地址。
- EIDRM：标识符为 msqid 的消息队列已被删除。
- EINVAL：无效的参数 cmd 或 msqid。
- EPERM：参数 cmd 为 IPC_SET 或 IPC_RMID，却无足够的权限执行。

例 9.11 是一个使用 msgctl 函数删除例 9.10 中创建的消息队列实例。

【例 9.11】使用 msgctl 删除消息队列

应用代码使用 argv 作为传递的键值参数，使用 atoi 函数将 argv[1]对应的字符串转换为对应的消息队列键值，然后调用 msgctl 函数来删除该消息队列。

实例的应用代码如下：

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <sys/types.h>
4  #include <sys/ipc.h>
5  #include <sys/msg.h>
6
7  int main(int argc,char *argv[])
8  {
9     int qid;
10    int ret;
11    if(argc < 2)
12    {
13       printf("请输入正确的键值\n");
14       exit(0);
15    }
16    qid = atoi(argv[1]);                    //获取键值
17    ret = msgctl(qid,IPC_RMID,NULL);    //删除消息队列
18    if(ret < 0)
19    {
20       printf("删除消息队列失败.\n");
21       exit(0);
```

```
22      }
23      else
24      {
25          printf("删除消息队列成功.\n");
26      }
27      return 0;
28  }
```

将文件保存为 exam911msgctl.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam911msgctl。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam911msgctl.c -o exam911msgctl
```

调用可执行文件 exam911msgctl 来删除指定的消息队列，首先使用 1627396137 作为键值，可以看到删除任务失败，然后再次使用“ipcs–q”命令获取的键值 0x61001829 作为参数，可以看到删除消息队列成功。再次调用“ipcs –q”命令来查看当前的消息队列，可以看到已经不存在该消息队列了。

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam911msgctl 1627396137
删除消息队列失败.
alloy@ubuntu:~/linuxc/chapter9$ ./exam911msgctl 0x61001829
删除消息队列成功.
alloy@ubuntu:~/linuxc/chapter9$ ipcs -q

--------- 消息队列 -----------
键          msqid        拥有者   权限       已用字节数  消息
```

3. 消息队列的发送和接收

Linux 提供了 msgsnd 函数用于消息队列的发送，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

如果函数调用成功，将返回“0”，否则返回“-1”，并且会设置对应的 error 参数值，对其详细说明如下。

- EAGAIN：参数 msgflg 设为 IPC_NOWAIT，而消息队列已满。
- EIDRM：标识符为 msqid 的消息队列已被删除。
- EACCESS：无权限写入消息队列。
- EFAULT：参数 msgp 指向无效的内存地址。
- EINTR：队列已满而处于等待情况下被信号中断。
- EINVAL：无效的参数 msqid、msgsz 或参数消息类型 type 小于 0。

Linux 内核提供了相应的函数 msgrcv 用于消息接收，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);
```

type 参数用来指定要接收消息队列中的哪条消息，并不是按照先进先出的顺序，其可分为三种情况，如表 9.6 所示。

表 9.6　type 参数的取值

参数取值	说明
0	返回消息队列中的第一个消息
>0	返回消息队列中的类型域等于这个值的第一个消息
<0	返回消息队列中类型域小于等于 type 绝对值的消息中，类型域最小的第一个消息

如果 msgrcv 函数调用成功，将返回接收的消息数据字节数，否则返回-1，同时将按照如下规则更新与消息队列 msqid 相连数据结构的成员：

- msg_qnum 减少 1。
- msg_lrpid 等于调用进程的进程 ID。
- msg_rtime 等于当前时间。

对 msgrcv 函数可能对应的 error 错误值说明如下：

- E2BIG：消息数据长度大于 msgsz，而 msgflag 没有设置 IPC_NOERROR。
- EIDRM：标识符为 msqid 的消息队列已被删除。
- EACCESS：无权限读取该消息队列。
- EFAULT：参数 msgp 指向无效的内存地址。
- ENOMSG：参数 msgflg 设为 IPC_NOWAIT，而消息队列中无消息可读。
- EINTR：等待读取队列内的消息时被信号中断。

例 9.12 和例 9.13 是一个进程使用消息队列进行数据发送和接收的应用实例（使用消息队列机制重新实现第 9.4.3 小节中的例 9.8 和例 9.9）。

【例 9.12】消息队列发送客户端

应用代码首先定义了一个名为 msgbuf 的信息结构体，然后使用“1234”作为键值调用函数 msgget 来创建一个消息队列，进入循环等待用户输入需要发送的字符串，将该字符串从标准输入从读入存放到消息队列中，并且调用 msgsnd 函数来发送该消息队列。在实现过程中应用代码设置了一个标志位 runningFlg 来作为退出消息队列的依据，当应用代码检测到用户从终端输入了“end”字符串，则修改该标志位的值以便退出循环，程序的流程如图 9.15 所示。

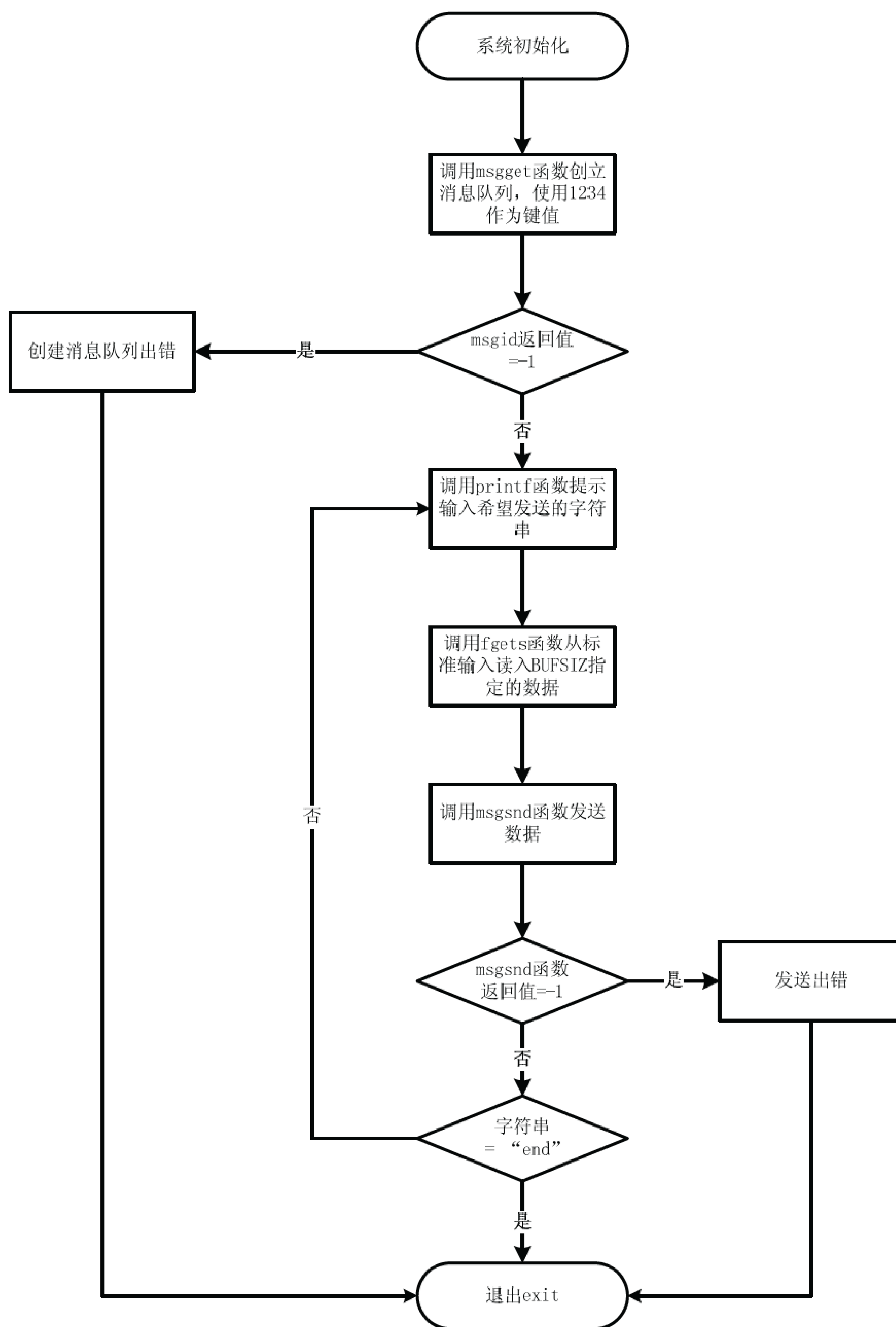


图 9.15　消息队列发送客户端工作流程

实例的应用代码如下：

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <unistd.h>
```

```
 5  #include <sys/types.h>
 6  #include <sys/ipc.h>
 7  #include <sys/msg.h>
 8  #include <error.h>
 9
10  //信息结构体
11  struct my_msg
12  {
13     long int my_msg_type;                    //数据类型
14     char text[BUFSIZ];                       //消息缓冲区的大小
15  } msgbuf;
16
17  int main(int argc,char *argv[])
18  {
19     int runningFlg =1;                       //运行标志
20     int msgid;                               //消息标识符
21     msgid = msgget((key_t)1234,0666 |IPC_CREATE);
       //创建一个消息队列，使用 1234 作为键值
22     if(msgid==-1)
23     {
24        perror("创建消息队列失败!\n");          //如果创建失败
25        exit(1);
26     }
27     while(runningFlg == 1)                   //如果程序处于运行中
28     {
29        printf("输入希望发送的字符串: ");
30        fgets(msgbuf.text,BUFSIZ,stdin);      //从标准输入读取 BUFSIZ 指定的数据
31        msgbuf.my_msg_type = 1;               //指定数据类型
32        if(msgsnd(msgid,(void *)&msgbuf, BUFSIZ, 0)==-1)             //发送数据
33        {
34           perror("发送消息失败!\n");                                //如果发送失败
35           exit(1);
36        }
37        if(strncmp(msgbuf.text,"end",3)==0)                          //如果用户输入 end
38        {
39           runningFlg = 0;     //结束运行
40        }
41      }
42      return 0;
43  }
```

将文件保存为 exam912msgqueuesnd.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam912msgqueuesnd。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam912msgqueuesnd.c -o exam912msgqueuesnd
```

【例 9.13】消息队列接收服务器端

应用代码首先调用 msgget 函数来建立消息队列，其键值依然是“1234”，然后进入循环等待接

收消息队列，将接收到的存放在 msgbuf.text 中的字符串通过 printf 函数打印输出。在循环中同样通过对 runningFlg 标志位的判断来决定是否跳出循环，和发送端不同的是，如果接收完成则调用 msgctl 函数来删除当前的消息队列，其流程如图 9.16 所示。

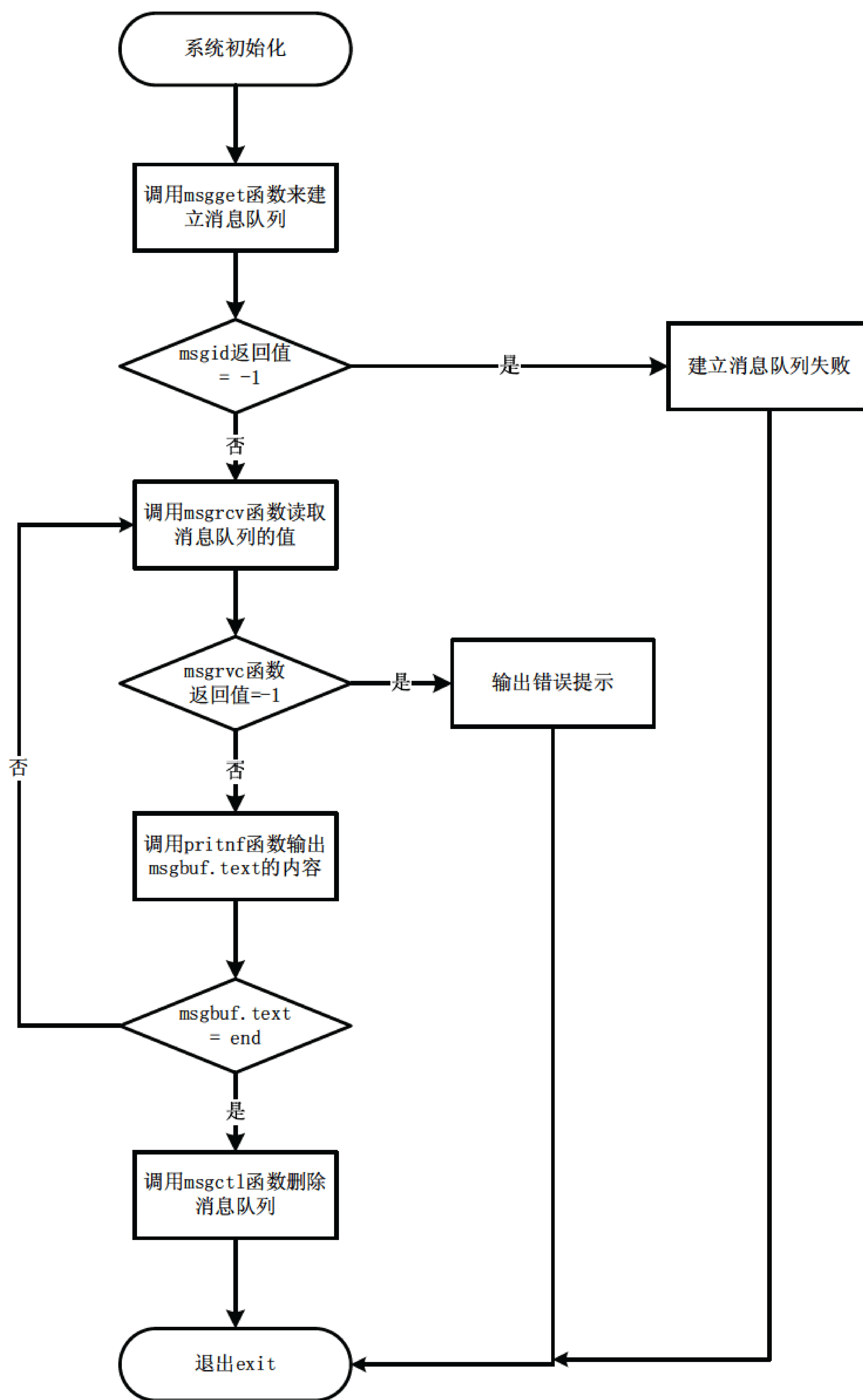


图 9.16　消息队列接收服务器端

实例的应用代码如下：

```
1   #include <stdlib.h>
2   #include <stdio.h>
```

```
 3  #include <string.h>
 4  #include <unistd.h>
 5  #include <sys/types.h>
 6  #include <sys/ipc.h>
 7  #include <sys/msg.h>
 8  #include <errno.h>
 9
10  //定义的消息队列的结构体
11  struct my_msg
12  {
13     long int my_msg_type;
14     char text[BUFSIZ];
15  } msgbuf;
16
17  int main(int argc,char *argv[])
18  {
19     int runningFlg =1;
20     int msgid;
21     long int msg_to_receive=0;
22     msgid = msgget((key_t)1234,0666 |IPC_CREATE);       //建立消息队列
23     if(msgid == -1)                                      //如果建立消息队列失败
24     {
25        printf("msgget failed!\n");
26        exit(1);
27     }
28     while(runningFlg == 1)                               //进入循环
29     {
30         if(msgrcv(msgid,(void *)&msgbuf, BUFSIZ,msg_to_receive, 0)==-1)
31         {
32             perror("msgrcv failed!\n");                  //如果接收数据失败
33             exit(1);
34         }
35         printf("接收到的字符串是 : %s", msgbuf.text);
36         if(strncmp(msgbuf.text,"end",3)==0)
37         runningFlg   = 0;                                //如果接收完成
38     }
39     if(msgctl(msgid, IPC_RMID, 0)==-1)                   //删除消息队列
40     {
41        perror("msgct(IPC_RMID)   failed!\n");            //如果删除消息队列失败
42        exit(1);
43     }
44     return 0;
45  }
```

将文件保存 exam912msgqueueget.c，在终端中使用 gcc 编译，生成可执行文件 exam912msgqueueget。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam912msgqueueget.c -o exam912msgqueueget
```

在两个终端中分别运行 exam912msgqueuesnd 和 exam912msgqueueget，并且发送相应的字符串，可以看到对应的发送和接收信息。

客户端：

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam912msgqueuesnd
输入希望发送的字符串: this is a test!
输入希望发送的字符串: hello
输入希望发送的字符串: 测试
输入希望发送的字符串:
```

服务器端：

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam912msgqueueget
接收到的字符串是 : this is a test!
接收到的字符串是 : hello
接收到的字符串是 : 测试
```

也可以同时利用“ipcs –q”命令来观察当前 Linux 下的消息队列状态。

9.7 Linux 的信号量

信号量（Semaphore）是一种用于实现计算机资源共享的 IPC 通信机制，其本质是一个计数器。

9.7.1 信号量的基础

1. 信号量的表现形式

信号量是用于控制多个进程访问计算机的共享资源机制，计算机的共享资源按照访问方法可以分为如下两类。

- 互斥共享：即某一时刻只能允许一个进程对资源进行操作。
- 同步共享：同一时刻可以由若干进程对其进行某种操作。

信号量就是用来帮助实现多进程对资源的共享机制，其名字来源于十字路口的信号灯。当红灯亮时，南北车辆通过路口，东西车辆等待；而绿灯亮时，东西车辆通过路口，南北车辆等待。由此，假设有某个共享资源，某一时刻只能允许一个进程对其进行操作，就像路口只能允许一个方向的车辆通过一样，是一种互斥资源。此时，信号量就像红绿灯一样，当某个进程对资源操作时，把它设置为一种形式，锁定资源，不允许其他进程使用；当这个进程完成操作后，释放资源，把它设置为另外一种形式，允许其他进程使用。这就是比较典型的信号量的使用形式，用于协调多个进程使用同一互斥资源。

信号量还有一种使用形式，用于处理多个共享资源。例如有六台打印机，若干人要使用。打印机总管手里有空闲打印机个数的记录：当有人要使用打印机时，总管查看空闲打印机数目记录，如果大于零，就可以将打印机资源分配出去，空闲打印机数减一，否则请使用者等待；使用者用完打印机后归还时，空闲打印机数加一，若有使用者等待，就将打印机分配出去。信号量在对多个共享

资源的控制中，就起到记录空闲资源数目的作用。

2. 信号量的基础定义

当 Linux 操作系统的一个进程要访问某个共享资源时，它按照下列步骤进行操作：

01 检测控制这个资源的信号量的值。

02 如果信号量的值是正数，就可以使用这个资源。进程将信号量的值减一，表示它正在使用资源的某个小单元。

03 如果信号量的值为零，那么这个进程进入睡眠状态，直到信号量的值重新大于零时被唤醒，转入第一步操作。

为了正确地实现信号量这一机制，检测和增减信号量的值应该是原语操作，所以信号量一般是在内核中实现的。

有一种信号量的普通形式，称为二元信号量。它只控制一个资源，信号量的初始值设为“1”。这就是前面介绍的十字路口信号灯的情况。普遍来讲，信号量的初值可以是任意的正整数，这个初值就是共享资源可以提供的可供共享的单元个数。

在信号量的实际应用中，是不能单独定义一个信号量的，而只能定义一个信号量集，其中包含一组信号量，同一信号量集中的信号量使用同一个引用 ID，这样的设置是为了多个资源或同步操作的需要。每个信号量集都有一个与之对应的结构，其中记录了信号量集的各种信息，该结构的定义如下，其定义位于头文件<sys/sem.h>中：

```
struct semid_ds
{
		struct ipc_perm		sem_perm;
		struct sem		*sem_base;
		ushort			sem_nsems;
		time_t			sem_otime;
		time_t			sem_ctime;
}
```

结构中分量的含义如表 9.7 所示。

表 9.7 结构 semid_ds 说明

分量	含义
sem_perm	与消息队列相同，该指针指向与信号量集相对应的 ipc_perm 结构的指针
sem_base	指向这个集合中第一个信号量的位置指针，这个域对于用户进程是没有用处的，实际指向一个 sem 结构的数组，数组中有 sem_nsems 个元素，每个元素对应信号量集中的一个信号量
sem_nsems	集合中信号量的个数
sem_otime	最近一次调用 semop 函数的时间
sem_ctime	最近一次改变的时间

下面介绍 semid_ds 中涉及的 sem 结构，这个结构中记录了单一信号量的一些信息，具体描述

如下：

```
struct sem
{
        ushort      semval;
        pid_t       sempid;
        ushort      semncnt;
        ushort      semzcnt;
}
```

其中每个分量的含义如表 9.8 所示。

表 9.8　结构 sem 中每个域的含义

分量	含义
semval	信号量的值
sempid	最近一次执行操作的进程的进程号
semncnt	等待信号值增长，即等待可利用资源出现的进程数
semzcnt	等待信号值减少到零，即等待全部资源可被独占的进程数

和消息队列相同，Linux 操作系统对于信号量集也有一些限制，如表 9.9 所示。

表 9.9　Linux 操作系统中的信号量集限制

名称	说明
SEMVMX	最大的信号值
SEMMNI	系统中允许的最大信号量集的个数
SEMMNS	系统中允许的最大信号量的个数
SEMMSL	每个信号量集中最大的信号量的个数

9.7.2　信号量的操作

Linux 内核提供了相应的函数对信号量进行相应的操作，这些操作包括创建或者打开一个信号量集、操作信号量集以及对信号量集进行控制。

1. 创建或者打开信号量集

semget 函数用于创建或者打开一个信号量集，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

参数 key 和参数 semflg 的用法可以参考第 9.6.2 小节中对 msgget 函数的介绍。nsems 参数用于指出信号量集中创建的信号量数量，如果是打开一个已存在的信号量集，这个参数就被忽略。

如果调用成功，则返回信号量集合标识符，否则返回“-1”，并且设置相应的 error 参数，对其

详细说明如下。

- EACCESS：没有权限。
- EEXIST：信号量集已经存在，无法创建。
- EIDRM：信号量集已经删除。
- ENOENT：信号量集不存在，同时 semflg 没有设置 IPC_CREATE 标志。
- ENOMEM：没有足够的内存创建新的信号量集。
- ENOSPC：超出限制。

当一个新的信号量集被创建时，与之相关联的 semid_ds 结构被初始化：

- ipc_perm 结构会被初始化，其中，mode 域的设置会按照 semflg 的要求进行。
- em_otime 被置为零。
- sem_ctime 被置为当前值。
- sem_nsems 被置为参数 nsems 的值。

2. 操作信号量集

Linux 操作系统提供了 semop 函数用于对信号量集进行操作，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

参数 semid 是一个通过 semget 函数返回的信号量集标识符；参数 nsops 标明了参数 sops 所指向数组中的元素个数；参数 sops 为 sembuf 结构的数组，其中每个元素表示一个操作，由于此函数是一个原子操作，一旦执行就将执行数组中所有的操作。

Linux 提供了一个结构 sembuf 用来说明 semop 函数要对信号量集执行的操作，其定义如下：

```
struct sembuf
{
    unsigned short   sem_num;
    short   sem_op;
    short   sem_flg;
}
```

在 sembuf 结构中，sem_num 是相对应的信号量集中的某一个资源（即指定将要进行操作的信号量），所以其值是一个从 0 到相应的信号量集的资源总数（ipc_perm.sem_nsems）之间的整数。sem_op 指明所要执行的操作，sem_flg 说明函数 semop 的行为。sem_op 的值是一个整数，对它的取值及所对应的操作如下说明。

- sem_op>0：表示进程对资源使用完毕，释放相应的资源数，并将 sem_op 的值加到信号量的值上。
- sem_op=0：进程阻塞直到信号量的相应值为 0，当信号量已经为 0，函数立即返回。如

果信号量的值不为 0，则依据 sem_flg 的 IPC_NOWAIT 位决定函数动作。sem_flg 指定 IPC_NOWAIT，则 semop 函数出错返回 EAGAIN。若 sem_flg 没有指定 IPC_NOWAIT，则将该信号量的 semncnt 值加 1，然后进程挂起直到下述情况发生：信号量的值为 0，将信号量的 semzcnt 的值减 1，函数 semop 成功返回；此信号量被删除（只有超级用户或创建用户进程拥有此权限），函数 smeop 出错返回 EIDRM；进程捕捉到信号，并从信号处理函数返回，在此情况下将此信号量的 semncnt 值减 1，函数 semop 出错返回 EINTR。

- sem_op<0：请求 sem_op 的绝对值资源。如果相应的资源数可以满足请求，则将该信号量的值减去 sem_op 的绝对值，函数成功返回。当相应的资源数不能满足请求时，则这个操作与 sem_flg 有关。若 sem_flg 指定 IPC_NOWAIT，则 semop 函数出错返回 EAGAIN。若 sem_flg 没有指定 IPC_NOWAIT，则将该信号量的 semncnt 值加 1，然后进程挂起直到下述情况发生：当相应的资源数可以满足请求，则该信号的值减去 sem_op 的绝对值，成功返回；此信号量被删除（只有超级用户或创建用户进程拥有此权限），函数 smeop 出错返回 EIDRM，进程捕捉到信号，并从信号处理函数返回，在此情况下将此信号量的 semncnt 值减 1，函数 semop 出错返回 EINTR。

如果函数调用成功则返回“0”，否则返回“-1”，其同样会设置 error 的对应值，对其详细说明如下。

- E2BIG：一次对信号量个数的操作超过了系统限制。
- EACCESS：权限不够。
- EAGAIN：使用了 IPC_NOWAIT，但操作不能继续进行。
- EFAULT：sops 指向的地址无效。
- EIDRM：信号量集已经删除。
- EINTR：当睡眠时接收到其他信号。
- EINVAL：信号量集不存在，或者 semid 无效。
- ENOMEM：使用了 SEM_UNDO，但无足够的内存用于创建所需的数据结构。
- ERANGE：信号量值超出范围。

3. 控制信号量集

Linux 同样提供了 semctl 函数用于对信号量集的控制，其被称为信号量控制函数，除了设置信号量初值之外，它还可以获取与信号量集合相连的数据结构 semid_ds，改变信号量集合拥有者以及访问权限，删除指定的信号量集合，查看与信号量集合有关的其他信息，如最后一个操作它的进程和在该信号量集合上等待的进程数等。

对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

semid 参数是信号量集的标识符，而 semnum 参数用于指定信号量集中的某一个信号量；cmd

用于指定具体的操作动作，如表 9.10 所示。

表 9.10　semctl 中的 cmd 参数说明

命令	说明
SETVAL	设置单个信号量的值
GETALL	返回信号量集合中所有信号量的值
SETALL	设置信号量集合中所有信号量的值
IPC_STAT	放置与信号量集合相连的 semid_ds 结构当前值与 arg.buf 指定的缓冲区
IPC_SET	用 arg.buf 指定结构值替代与信号量集合相连的 semid_ds 结构值
GETVAL	返回单个信号量的值
GETPID	返回最后一个操作该信号量集合的进程 ID
GETNCNT	返回 semncnt 之值
GETZCNT	返回 semzcnt 之值
IPC_RMID	删除指定的信号量集合

执行 IPC_SET、IPC_RMID 命令的进程只能是信号量集合的创建进程、拥有进程或特权进程，执行其他命令的进程必须拥有信号量集合的读取或更新权限。

根据实际的 cmd 参数的具体内容，semctl 可能拥有第 4 个参数 arg，其是一个联合体（union），val 用于 SETVAL 命令，指明要设置的信号量值；buf 用于 IPC_STAT/IPC_SET 命令，表示存放信号量集合数据结构的缓冲区；array 用于 GETALL/SETALL 命令，存放所获得的或要设置的信号量集合中所有信号量的值，对其说明如下：

```
union semun
{
        int val;
        struct semid_ds *buf ;
        unsigned short array;
};
```

如果 semctl 函数调用成功，则返回值大于等于 0（当 semctl 的操作为 GET 操作时返回相应的值，其余返回 0）；如果调用失败则返回“－1”，并设置错误变量 errno 为对应的值，对其详细说明如下。

- EACCESS：权限不够。
- EFAULT：arg 指向的地址无效。
- EIDRM：信号量集已经删除。
- EINVAL：信号量集不存在，或者 semid 无效。
- EPERM：进程有效用户没有 cmd 的权限。
- ERANGE：信号量值超出范围。

9.8　Linux 的共享内存

共享内存是指让 Linux 操作系统中两个或者更多进程共享一段指定的内存区间进行数据交互的 System V IPC 机制，这是三种 IPC 机制中速度最快的一种。

9.8.1　共享内存的基础

两个不同进程 A、B 共享内存的意思是：同一块物理内存被映射到进程 A、B 各自的进程地址空间，进程 A 可以即时看到进程 B 对共享内存中数据的更新，反之，进程 B 也可以即时看到进程 A 对共享内存中数据的更新。

共享内存机制是最快的一种进程通信机制，因为没有中间介质，如消息队列、管道等的延迟，数据直接由内存映射到进程空间。通常，共享内存段由一个进程创建，接下来的读写操作就由许多进程参加，这样就能传递信息了。

在系统内核为一个进程分配内存地址时，通过分页机制可以让一个进程的物理地址不连续，同时也可以让一段内存同时分配给不同的进程。共享内存机制就是通过该原理来实现的，共享内存机制只是提供数据的传送，如何控制服务器端和客户端的读写操作互斥，这就需要一些其他的辅助工具，例如信号量的概念。

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行 4 次数据拷贝，而共享内存则只拷贝 2 次数据：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，在进程之间共享内存时，并不总是读写少量数据后就解除映射，当有新的通信时，再重新建立共享内存区域，而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的，因此，采用共享内存的通信方式效率是非常高的。

共享内存机制的唯一不足在于，需要一定的同步机制控制多个进程对同一块内存的读写。当一个进程在写数据时，不允许其他的进程写数据或读数据，这可以通过信号量的控制实现。

同前面的两种通信机制一样，每个共享内存段都对应一个 shmid_ds 结构。这个结构的定义如下：

```
struct shmid_ds
{
        struct ipc_perm   shm_perm;
        int               shm_segsz;
        ushort            shm_lkcnt;
        pid_t             shm_cpid;
        pid_t             shm_lpid;
        ulong             shm_nattach;
        time_t            shm_atime;
        time_t            shm_dtime;
        time_t            shm_ctime;
};
```

其中每个分量的含义如表 9.11 所示。

表 9.11　shmid_ds 结构分量说明

分量	含义
shm_perm	与信号量相同，这个指针指向与这个共享内存相对应的 ipc_erm 结构指针
shm_segsz	共享内存段的大小，以字节计
shm_lkcnt	共享内存段被锁定的时间数
shm_lpid	最近一次调用 shmop 函数的进程的进程号
shm_cpid	创建这个共享内存段的进程的进程号
shm_nattach	当前把这个内存段附加到地址空间的进程数
shm_atime	最近一次附加操作的时间
shm_dtime	最近一次分离操作的时间
shm_ctime	最近一次改变的时间

和信号量、消息队列类似，Linux 同样给共享内存提供了一些限制，如表 9.12 所示。

表 9.12　Linux 操作系统下的共享内存限制

名称	说明
SHMMAX	共享内存段的最大字节数
SHMMIN	共享内存段的最小字节数
SHMMNI	系统中允许存在的共享内存的最大个数
SHMSEG	一个进程中允许存在的共享内存的最大个数

9.8.2　共享内存的操作

Linux 内核提供了一系列函数用于对共享内存进行操作，这些操作包括共享内存的创建或打开、共享内存的连接、共享内存的脱离以及共享内存的属性设置。

1．创建或者打开共享内存

shmget 函数用于创建一块新的共享内存或者打开一块已经存在的内存，对其标准调用格式说明如下：

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

参数 key 表示所创建或打开的共享内存的关键字；参数 size 表示共享内存区域的大小，只在创建一个新的共享内存时生效；参数 shmflg 表示调用函数的操作类型，也可用于设置共享内存的访问权限，两者通过逻辑或表示。

shmget 函数调用成功时，返回值为共享内存的引用标识符；调用失败时，返回值为“-1”，并且设置相应的 error 值，详细说明如下。

- EINVAL：参数 size 小于 SHMMIN 或大于 SHMMAX。
- EEXIST：预建立 key 所指的共享内存，但已经存在。
- EIDRM：参数 key 所指的共享内存已经删除。
- ENOSPC：超过了系统允许建立的共享内存的最大值（SHMALL）。
- ENOENT：参数 key 所指的共享内存不存在，而参数 shmflg 未设 IPC_CREATE 位。

- EACCES：没有权限。
- ENOMEM：核心内存不足。

函数 shmget 的具体动作由参数 key 和 flag 决定，对其详细说明如下。

- 当 key 参数被设置为 IPC_PRIVATE 时，创建一个新的共享内存。此时参数 shmflg 的取值对函数的操作不起任何作用。
- 当 key 参数没有被设置为 IPC_PRIVATE，且参数 shmflg 设置了 IPC_CREATE 位，而没有设置 IPC_EXCL 位，则执行操作由 key 取值决定；另外如果 key 参数为内核中某个已存在的共享内存的键，则执行打开这个键的操作；反之，执行创建共享内存的操作。
- 当 key 参数没有被设置为 IPC_PRIVATE，且参数 shmflg 中同时设置了 IPC_CREATE 位和 IPC_EXCL 位，则只执行创建共享内存操作。参数 key 的取值应与内核中已存在的任何共享内存的关键字都不相同，否则函数调用失败。

当调用 shmget 函数创建一个共享内存时，此共享内存的 shmid_ds 结构被初始化，对其初始化规则说明如下。

- ipc_perm：各个分量被设置为相应值。
- shm_lpid、shm_nattach、shm_atime 和 shm_dtime：被设置为 0。
- shm_ctime：设置为当前时间。

2. 连接共享内存

当一个共享内存创建或打开后，使用该共享内存的进程必须将此内存区域附加到它的地址空间，Linux 提供了 shmat 函数用于连接共享内存，对其标准调用格式说明如下。

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

参数 shmid 表示要附加的共享内存段的引用标识符；参数 shmflg 用于表示 shmat 函数的操作方式，如果 shmflg 设置了 SHM_RDONLY 位，则该内存区域被设置为只读，否则设置为可读写；参数 shmaddr 和参数 shmflg 共同决定共享内存区域要附加到的地址值，对其详细说明如下：

- 如果参数 shmaddr 为 0，系统将自动查找进程地址空间、将共享内存区域附加到第一块有效内存区域上，此时 shmflg 无效。
- 如果参数 shmaddr 不为 0，而参数 shmflg 未设置 SHM_RND 位，则共享内存区域附加到由 shmaddr 指定的地址处。
- 如果参数 shmaddr 不为 0，而参数 shmflg 设置了 SHM_RND 位，则共享内存区域附加到由 shmaddr-(shmaddr %SHMLBA)指定的地址处。

如果 shmat 函数调用成功，则返回共享内存区域的指针；若调用失败，则返回值为“-1”，同时会设置 error 参数，对其详细说明如下：

- EACCES：无权限以指定方式连接共享内存。
- EINVAL：无效的参数 shmid 或 shmaddr。
- ENOMEM：核心内存不足。


注 意

shmat 函数成功执行后，会将 shmid 所表示共享内存段的 shmid_ds 结构的 shm_nattch 计数器的值加 1。

3. 脱离共享内存

在使用完共享内存之后，应该调用 shmdt 函数将指定的共享内存段从当前进程空间中脱离出去，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

shmdt 函数仅用于将共享区域与进程的地址空间分离，并不删除共享内存本身。参数 shmaddr 为要分离的共享内存区域的指针，是调用 shmdt 函数时的返回值。

若函数调用成功，则返回“0”，如果调用失败，则返回“-1”，并且设置相应的 error 值为 EINVAL，以表示无效的参数 shmaddr。

4. 设置共享内存的属性

Linux 同样提供了对共享内存进行控制的函数 shmctl，对其标准调用格式说明如下；

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

参数 shmid 为所要操作的共享内存段的标识符；参数 buf 是 struct shmid_ds 型的指针，其作用与参数 cmd 相似；参数 cmd 用于指明 shmctl 函数所要进行的操作，如表 9.13 所示。

表 9.13　参数 cmd 的设置

cmd 的值	含义
IPC_STAT	取 shmid 所指向内存共享段的 shmid_ds 结构，对参数 buf 指向的结构赋值
IPC_SET	使用 buf 指向的结构对 shmid 段的相关结构赋值，只对以下几个域有作用：shm_perm.uid、shm_perm.gid 以及 shm_perm.mode
IPC_RMID	删除 shmid 所指向的共享内存段，只有当 shmid_ds 结构的 shm_nattch 域为零时，才会真正执行删除命令，否则不会删除该段。注意此命令的请求规则与 IPC_SET 命令相同
SHM_LOCK	锁定共享内存段，此命令只能由超级用户请求
SHM_UNLOCK	对共享内存段解锁，此命令只能由超级用户请求


注 意

IPC_SET 参数要求进程的用户 ID 等于 shm_perm.cuid 或者等于 shm_perm.uid，又或者拥有 root 权限。

如果函数 shmctl 调用成功则返回“0”，如果出错则返回“-1”，并且设置 error 为对应的值，对其详细说明如下。

- EACCESS：参数 cmd 为 IPC_STAT，无权限读取该共享内存。
- EFAULT：参数 buf 指向无效的内存地址。
- EIDRM：标识符为 msqid 的共享内存已被删除。
- EINVAL：无效的参数 cmd 或 shmid。
- EPERM：参数 cmd 为 IPC_SET 或 IPC_RMID，无足够的权限执行。

9.8.3 共享内存的应用实例

例 9.14 是一个使用信号量控制共享内存同步以实现父进程和子进程进行数据通信的实例，对其实现步骤说明如下。

01 应用代码创建信号量，映像共享内存。
02 创建子进程，此时子进程继承了父进程的所有上下文。
03 调用 sleep 函数使得父进程睡眠 1 秒，从而让子进程首先运行。
04 子进程运行，立刻初始化信号量并且申请访问共享资源。
05 子进程调用 sleep 函数睡眠 4 秒。
06 此时父进程运行，但信号量已为 0，所以父进程被阻塞。
07 子进程接着运行，最后释放共享内存。
08 父进程被换醒，获得信号锁运行，完成对应的操作后删除共享内存退出。

【例 9.14】使用信号量和共享内存进行父子进程通信

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/ipc.h>
4   #include <sys/shm.h>
5   #include <sys/types.h>
6   #include <unistd.h>
7   #include <string.h>
8   #include <sys/sem.h>
9   #define SHM_SIZE 1024                          //共享内存的大小
10  int main(int argc,char *argv[])
11  {
12      int ret,                                   //临时变量
13      pid,                                       //进程 id
14      sme_id,                                    //保存信号量描述符
```

```
15      shm_id;                          //保存共享内存描述符
16      key_t       sme_key,             //保存信号量键值
17      shm_key;                         //保存共享内存键值
18      char *shmp;                      //指向共享内存的首地址
19      struct shmid_ds dsbuf;           //定义共享内存信息结构变量
20      struct sembuf lock = {0, -1, SEM_UNDO};                   //信号量上锁操作的数组指针
21      struct sembuf unlock = {0, 1, SEM_UNDO | IPC_NOWAIT};     //信号量解锁操作的数组指针
22      shm_key = ftok(*(argv+1), 2);                             //获取信号量键值
23      if(shm_key < 0)
24      {
25          perror("ftok");                                       //调用 ftok 函数出错
26          exit(0);
27      }
28      sme_id = semget(shm_key, 1, IPC_CREATE | 0666);           //获取信号量 ID
29      if(sme_id < 0)
30      {
31          perror("semget");                                     //调用 semget 函数出错
32          exit(0);
33      }
34      shm_key = ftok(*(argv+2), 1);                             //获取共享内存键值
35      if(shm_key < 0)
36      {
37          perror("ftok");
38          exit(0);
39      }
40      shm_id = shmget(shm_key, SHM_SIZE, IPC_CREATE | 0666);    //获取共享内存 ID
41      if(shm_id < 0)
42      {
43          perror("shmget");
44          exit(0);
45      }
46      shmp = shmat(shm_id, NULL, 0);                            //映像共享内存
47      if((int)shmp == -1)
48      {
49          perror("shmat");
50          exit(0);
51      }
52      pid = fork();                                             //创建子进程
53      if(pid < 0)
54      {
55          perror("fork");
56          exit(0);
57      }
58      else if(pid == 0)                                         //子进程
59      {
60          ret = semctl(sme_id, 0, SETVAL, 1);                   //初始化信号量，初值设为 1
61          if(ret == -1)
62          {
63              perror("semctl");
64              exit(0);
65          }
```

```
66          ret = semop(sme_id, &lock, 1);                    //申请访问共享资源，锁定临界资源
67          if(ret == -1)
68          {
69             perror("semop lock");
70             exit(0);
71          }
72          sleep(4);                                         //让子进程睡眠 4 秒
73          strcpy(shmp, "hello\n");                          //向共享内存写入数据
74          if(shmdt((void *)shmp) < 0)                       //使共享内存脱离进程地址空间
75          {
76             perror("shmdt");
77          }
78          ret = semop(sme_id, &unlock, 1);                  //解锁临界资源
79          if(ret == -1)
80          {
81             perror("semop unlock");
82             exit(0);
83          }
84       }
85       else                     //父进程
86       {
87          sleep(1);             //先让子进程运行
88          ret = semop(sme_id, &lock, 1);                    //申请访问共享资源，锁定临界资源
89          if(ret == -1)
90          {
91             perror("semop lock");
92             exit(0);
93          }
94          if(shmctl(shm_id, IPC_STAT, &dsbuf) < 0)          //获取共享内存信息
95          {
96             perror("shmctl");
97             exit(0);
98          }
99          else                                              /* 共享内存的状态信息获取成功 */
100         {
101            printf("Shared Memory Information:\n");
102            printf("\tCreateor PID: %d\n", dsbuf.shm_cpid);   /*输出创建共享内存进程的标识符 */
103            printf("\tSize(bytes): %d\n",dsbuf.shm_segsz);    /* 输出共享内存的大小 */
104            printf("\tLast Operator PID: %d\n",dsbuf.shm_lpid);
               /* 输出上一次操作共享内存进程的标识符 */
105            printf("Received message : %s\n", (char *)shmp);      /* 从共享内存中读取数据 */
106         }
107         if(shmdt((void *)shmp) < 0)                       //使共享内存脱离进程地址空间
108         {
109            perror("shmdt");
110            exit(0);
111         }
112         ret = semop(sme_id, &unlock, 1);              //解锁临界资源
113         if(ret == -1)
114         {
115            perror("semop unlock");
```

```
116             exit(0);
117         }
118         if(shmctl(shm_id, IPC_RMID, NULL) < 0)        /* 删除前面创建的共享内存 */
119         {
120             perror("shmctl");
121             exit(0);
122         }
123         ret = semctl(sme_id, 0, IPC_RMID, NULL);     //删除信号量
124         if(ret == -1)
125         {
126             perror("semctl");
127             exit(0);
128         }
129     }
130     return 0;
131 }
```

将文件保存为 exam913semandmem.c，在终端使用 gcc 进行编译链接，生成可执行文件 exam913semandmem。

```
alloy@ubuntu:~/linuxc/chapter9$ gcc exam913semandmem.c -o exam913semandmem
```

分别使用例 9.7 中创建的文件 myfifotest 和例 9.8 中使用的文件 MYFIFO 来创立键值，可以看到如下的输出：

```
alloy@ubuntu:~/linuxc/chapter9$ ./exam913semandmem MYFIFO myfifotest
Shared Memory Information:
        Createor PID: 3190
        Size(bytes): 1024
        Last Operator PID: 3191
Received message : hello
```

9.9　本章习题

1. 假设存在一个可以从标准输入读入字母并且将其从小写转换为大写输出的可执行程序 upcase，使用 pipe 函数编写一个应用程序，用于实现将一个指定文本文件中的字母转换为大写。

2. 使用 popen 函数来重写上一题的应用程序。

3. 编写一个程序，使用 pipe 函数创建一个匿名管道，并使用 write 向管道的一端写入数据，使用 read 函数从管道的另一端读取数据。

4. 编写一个程序，使用 msgget 函数创建一个消息队列，并返回该消息队列的描述符。

5. 编写一个程序，使用 msgsnd 函数向消息队列中发送一个字符串数据信息“Hello!This is a test!”，并通过查看消息队列的属性信息检验发送是否成功。

6. 编写一个程序，使用 semget 函数创建一个信号量集，并返回该信号量集的描述符。

7. 编写一个程序，使用 write 函数向共享内存中写入数据，实现不同进程间的数据信息传递。

第 10 章　Linux 的线程

线程，有时也被称为轻量级进程（Light Weight Process，LWP），是程序执行流的最小单元，本章将介绍 Linux 下线程的基础知识，涉及以下内容：

- Linux 下线程的基础知识。
- Linux 下线程的基础操作方法，包括创建线程、阻塞线程、分离线程、阻塞和清理线程以及如何使得线程退出。
- Linux 线程中私有数据的处理方法。
- Linux 线程的属性。

10.1　Linux 的线程基础

线程的处理是 Linux 的 C 语言编程中相对高级一些的内容，在学习对其进行操作的方法之前应该首先了解线程的一些特点。

10.1.1　线程的特点

前面介绍的典型 Linux 进程可以看成其只有一个控制线程，所以这个进程在同一时刻只能做一件事情，如果在一个应用程序中存在多个进程，则有一些明显的缺点：

- 由于 fork 是一个花销很大的系统调用，所以创建这些进程增加了一些基本的开销，这是因为启动一个新的进程必须分配给其独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段。
- 由于每个进程都有自己的地址空间，因此必须使用进程间通信的手段，如管道共享内存。
- 要把这些进程分配到不同的机器或处理器上运行，以及在进程之间传递信息、等待进程的完成、收集结果等都需要额外的开销。

如果使用线程则可以使得这个进程在“同一时刻”能够同时完成多个任务，这种方式有如下的优点：

- 提高应用程序响应：这对图形界面的程序尤其重要，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作（time consuming）置于一个新的线程，可以避免这种尴尬的情况。
- 使多处理器系统更加有效：操作系统会保证当线程数不大于处理器数目时，不同的线程运行于不同的处理器上。
- 改善程序结构：一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序将有利于理解和修改。

一个标准的线程由线程标识符、当前指令指针（PC）、寄存器集合和堆栈等组成，其是进程中的一个实体，是被系统独立调度和分派的基本单位。线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源，但它可与同属一个进程的其他线程共享进程所拥有的全部资源。

线程包含了表示进程内执行环境必须的信息，这些信息包括线程标识符（线程 ID）、一组寄存器值、栈、调度优先级和策略、信号屏蔽字、errno 变量以及线程私有数据。进程的所有信息对该进程的所有线程都是共享的，包括可执行的程序文本、程序的全局内存和堆内存、栈和文件描述符。

10.1.2　控制线程和线程的标识符

在 Linux 创建新的线程时，其会有一个控制线程用于控制新线程的相应工作，被称为主线程或者控制线程，如图 10.1 所示。

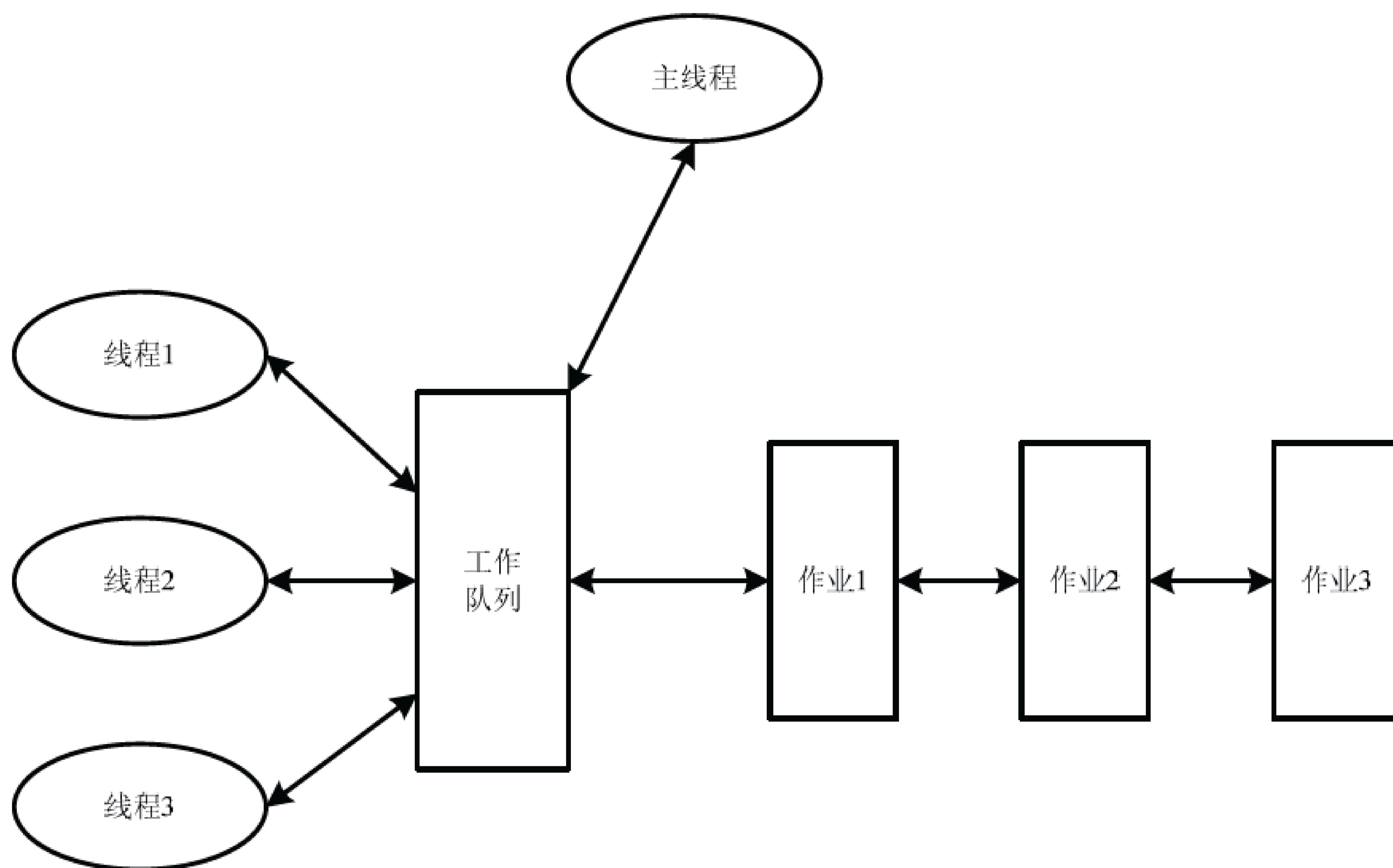


图 10.1　主线程/控制线程

和进程标识符类似，每一个线程都有一个在进程中唯一的线程标识符（线程 ID），其用一个数据类型 pthread_t 来表示，其实该数据类型在 Linux 中就是一个无符号长整型数据。

Linux 提供了两个函数用于对线程标识符的操作，对其标准调用格式说明如下：

```
#include <pthread.h>
pthread_t    pthread_self(void);
```

pthread_self 函数用于获得线程自身的线程标识符，其返回值是线程自身的线程标识符。

pthread_equal 函数用于比较两个线程标识符，对其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

函数的两个参数分别是需要比较的两个线程标识符，如果相等则返回一个非“0”值，否则返回“0”值。

10.1.3　用户态和核心态线程

用户态线程在管理上不需要内核的参与，所以通常又称为“协作式多任务”，在进程内的这些线程统一由用户程序来切换，所以每一个线程在执行完任务后，调用任务切换功能，并向其发送信号，任务切换完成。线程对处理器资源的占用也切换到其他线程。通常情况下，用户态线程在线程切换时要比内核线程的速度快，不过在几个比较成功的内核态线程库中，线程切换的速度也相当快。虽然用户态线程有灵活和快速的特性，但是也存在一个严重的问题，即进程中的一个线程可能独占整个时间片，导致其他线程得不到处理器时间而无法运行，例如，当一个线程由于磁盘 I/O 而阻塞时，其他线程同样也不能运行。另外，用户态线程不能发挥多处理器机器（SMP）的性能。

内核态线程是由内核来管理的，在每一个时间片内，内核负责调度进程内的线程。由于内核参与了用户态进程的调度，所以就涉及了内核态与用户态上下文的切换。通常所说的内核态线程切换速度慢就是由于这个原因导致的。但是使用内核态线程的一个明显的好处是进程内的一个线程不会独占整个进程的处理器时间，这样，如果一个线程由于磁盘 I/O 而阻塞，其他线程仍可以利用处理器时间运行。使用核心态线程的另外一个好处是可以充分发挥 SMP 系统的性能，而且随着系统处理器数量的增多，应用程序运行的速度明显加快。

10.1.4　使用 gcc 编译线程的相关代码

利用 gcc 编译多线程程序时，必须与 pthread 函数库连接。在终端编译时可使用下列命令：

```
gcc –lpthread
```

上述编译命令把程序与 pthread 函数库相连。

10.2　Linux 的线程操作

线程的操作包括线程的创建、退出和终止、阻碍和分离、取消和清理等，其和进程的操作比较如表 10.1 所示，本小节将详细介绍这些相应的操作。

表 10.1　线程和进程操作函数比较

进程函数	线程函数	说明
fork	pthread_create	创建一个线程或者进程
exit	pthread_exit	退出线程或者进程
waitpid	pthread_join	处理进程或者线程退出之后的状态
atexit	pthread_cleanup_push	退出控制流所调用的函数
getpid	pthread_self	获得标识符
abort	pthread_cancel	控制线程或者进程退出

10.2.1 创建线程

在 Linux 中，可以调用 pthread_create 函数创建一个新的线程，对其标准调用格式说明如下；

```
#include <pthread.h>
int pthread_create (pthread_t *thread, pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
```

对函数的各个参数说明如下，如果创建成功则返回“0”，否则返回错误编号。

- 参数 thread：线程的标识符，需要说明的是这个参数并不是由用户所确定的，用户只需要声明一个 pthread_t 类型的数据变量，并且将其传递给 pthread_create 函数，函数在创建新的线程时会将新的线程标识符放到这个变量中。
- 参数 attr：指定线程的属性，其详细信息将在第 10.3 节中介绍，也可以将其设置为 NULL。
- 参数 start_routine：用于指定开始运行的函数，新创建的线程是从这个函数开始运行的，用户需要指定这个函数。
- 参数 arg：这是函数 start_routine 所需要的参数，这是一个无类型指针，如果需要传递的参数不止一个，则需要将这些参数都放到一个结构中，然后将这个结构的地址传给 arg。


注 意

pthread_create 函数在调用失败之后会返回对应的错误编码，每个线程都会提供 errno 的副本。

例 10.1 是使用 pthread_create 函数来创建一个线程的实例。

【例 10.1】使用 pthread_create 函数来创建一个线程

应用代码首先定义一个函数 threaddeal，其是线程的入口函数，进入线程之后会首先执行该函数，在该函数中调用 printf 函数打印输出一个字符串；在主程序中调用 pthread_create 函数来创建一个新线程，使用 threadid 作为线程的标识符参数，使用 threaddeal 作为线程的入口函数参数，线程的参数和函数传递参数都设置为 NULL。

实例的应用代码如下：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  //新线程首先运行的函数
5  void *threaddeal(void *arg)
6  {
7     printf("这是一个新线程.\n");                          //输出新线程提示
8  }
9  int main(int arg,char *argv[])
10 {
11    pthread_t threadid;                                  //线程的标识符
12    if(pthread_create(&threadid,NULL,threaddeal,NULL) != 0)
13    //创建一个新线程，然后运行 threaddeal 函数
14    {
```

```
15          //如果返回值不是 0，则表示创建线程失败
16          printf("%s 错误出现在第%s 行",__FUNCTION__,__LINE__);   //打印错误信息
17          exit(0);
18      }
19      else
20      {
21          sleep(1);       //挂起 1 秒等待线程运行
22      }
23      return 0;
24  }
```

将文件保存为 exam101createthread.c，在终端使用 gcc 进行编译链接，生成可执行文件，需要注意的是最后必须加上“-lpthread”参数，否则会有找不到线程操作函数的错误。

```
alloy@ubuntu:~/linuxc/chapter10$ gcc exam101createthread.c -o exam101createthread -lpthread
```

运行该可执行文件，可以看到对应的输出字符串。

```
alloy@ubuntu:~/linuxc/chapter10$ ./exam101createthread
这是一个新线程.
```



注 意

在例 10.1 中没有让线程退出，这在实际应用中是不妥的，在第 10.2.2 小节中将介绍线程的退出方法。

在例 10.1 的创建线程实例中没有使用 pthread_create 函数的 arg 参数，这个参数是用于给线程入口函数传递数据的，例 10.2 给出了一个使用该参数的实例。

【例 10.2】使用 pthread_create 函数的 arg 参数

应用代码在主程序中制定了一个循环 10 次的 for 循环，每次将 for 循环的计数参数通过 arg 参数传递给线程的入口函数 threaddeal，然后在其中打印当前的 i 值。

实例的应用代码如下：

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <pthread.h>
 4  //线程处理函数
 5  void *threaddeal(void *arg)
 6  {
 7      printf("%d\n",*((int *)arg));                    //传递线程的参数
 8      pthread_exit(NULL);
 9  }
10
11  int main(int argc,char *argv[])
12  {
13      int i;
14      pthread_t threadid;
15      for(i=0;i<10;i++)
```

```
16      {
17          if(pthread_create(&threadid,NULL,threaddeal,&i) != 0)          //将 i 值作为参数传递
18          {
19              //返回值不为 0，则表明创建线程失败
20              printf("创建线程失败.\n");
21              exit(0);        //退出
22          }
23      }
24      pthread_exit(NULL);
25      return 0;
26  }
```

将文件保存为 exam102threadargv.c，在 gcc 中编译链接，生成可执行文件 exam102threadargv。

```
alloy@ubuntu:~/linuxc/chapter10$ gcc exam102threadargv.c -o exam102threadargv -lpthread
```

执行该可执行文件，可以看到依次打印输出计数器的值。

```
alloy@ubuntu:~/linuxc/chapter10$ ./exam102threadargv
1
2
3
7
6
4
7
8
9
0
```


注 意

在例 10.2 中调用了 pthread_exit 函数使得线程退出，以下分别是删除了处理函数 threaddeal 中的线程退出语句以及主程序中的线程退出语句之后的输出，读者可以自行研究导致这种错误的原因。

这是没有主程序中的线程退出语句的代码运行输出：

```
alloy@ubuntu:~/linuxc/chapter10$ ./exam102threadargv
2
2
3
4
5
6
7
8
32715
```

这是没有 threaddeal 函数中的线程退出语句的代码运行输出：

```
alloy@ubuntu:~/linuxc/chapter10$ ./exam102threadargv
2
3
8
4
7
7
3
9
10
0
```

10.2.2 阻塞和退出线程

当一个线程已经执行完成之后，可以被其他的线程阻塞挂起，然后等待指定的线程调用 pthread_exit，以便从启动例程中返回或者被取消，Linux 内核可以调用 pthread_join 函数来完成对线程的阻塞，对其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

参数 thread 是一个线程标识符，用于指定要等待其终止的线程；参数 retval 用于存放其他线程的返回值，对于每一个可连接的线程都必须调用该函数一次。任何线程都不能对相同的线程调用此函数，如果调用成功，则函数返回 0，否则返回一个非零值。

进程可以调用 exit 系列函数退出当前进程，线程也可以通过如下三种方式退出，在不终止整个进程的情况下停止线程的控制流。

- 线程只是从启动例程中返回，返回值是线程的退出码。
- 线程可以被同一个进程中的其他线程终止。
- 线程调用 pthread_exit 函数退出。

Linux 内核提供了 pthread_exit 函数用于主动退出线程，对其标准调用格式说明如下：

```
#include <pthread.h>
void pthread_exit(void *retval);
```

pthread_exit 函数没有返回值，其参数 retval 是线程的终止状态，其与 pthread_create 函数的 start_routine 参数类似，都是由用户先指定并且传递给函数的一个参数，在 pthread_exit 函数完成之后可以调用这个参数，以获得进程的退出状态。

例 10.3 是一个线程的阻塞和退出实例。

【例 10.3】线程的阻塞和退出

应用代码首先调用 pthread_create 函数创建一个线程，使用 threaddeal 作为线程的入口函数，在其中打印输出一个字符串，然后调用 pthread_exit 函数退出线程，并且使用“pthread exit”作为函数的参数；在主程序中使用 pthread_join 函数等待线程退出，此时线程的返回值（pthread exit）

存放到了 pthread_join 函数的参数 str 中，将其打印输出。

实例的应用代码如下：

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <stdlib.h>
4  //这是线程处理函数
5  void *threaddeal(void *arg)
6  {
7    printf("这是一个线程处理函数.\n");
8    pthread_exit("pthread exit");                              //线程退出
9  }
10 int main(int argc,char *argv[])
11 {
12   pthread_t threadid;
13   void *str;
14   if(pthread_create(&threadid,NULL,threaddeal,NULL) != 0)    //创建线程
15   {
16     //创建线程失败
17     printf("创建线程失败.\n");
18     exit(0);
19   }
20   else    //创建线程成功
21   {
22     pthread_join(threadid,&str);                             //等待新线程结束
23     printf("%s\n",(char *)str);                              //输出线程的退出状态
24   }
25   return 0;
26 }
```

将文件保存为 exam103threadquit.c，在终端使用 gcc 进行编译链接，生成可执行文件。

```
alloy@ubuntu:~/linuxc/chapter10$ gcc exam103threadquit.c -o exam103threadquit -lpthread
```

运行可执行文件，可以看到对应的输出，第一个汉字字符串是线程入口函数 threaddeal 中的输出，第二个字符串则是线程退出时的返回值。

```
alloy@ubuntu:~/linuxc/chapter10$ ./exam103threadquit
这是一个线程处理函数.
pthread exit
```

例 10.4 是一个多线程阻塞和退出的应用实例。

【例 10.4】多线程的阻塞和退出

应用代码在主程序中使用 pthread_create 函数创建两个线程，这两个线程分别对应的入口函数为 threaddeal1 和 threaddeal2，它们的退出值分别是 1 和 2，在主函数中分别调用 pthread_join 函数阻塞线程 1 和线程 2 等待它们退出，然后打印这两个线程的退出值。

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <pthread.h>
3   #include <stdlib.h>
4   //线程 1 的启动函数
5   void *threaddeal1(void *arg)
6   {
7      printf("thread 1 returning\n");
8      return((void *)1);
9   }
10  //线程 2 的启动函数
11  void *threaddeal2(void *arg)
12  {
13      printf("thread 2 exiting\n");
14      pthread_exit((void *)2);
15  }
16  //主函数
17  int   main(int argc,char *argv[])
18  {
19      int                 err;
20      pthread_t           threadid1, threadid2;
21      void                *tret;
22      //创建线程 1
23      err = pthread_create(&threadid1, NULL, threaddeal1, NULL);
24      if (err != 0)    //创建线程 1 失败
25      {
26         printf("创建线程 1 失败，错误为: %s\n", strerror(err));
27      }
28      //创建线程 2
29      err = pthread_create(&threadid2, NULL, threaddeal2, NULL);
30      if (err != 0)
31      {
32         printf("创建线程 2 失败，错误为: %s\n", strerror(err));
33      }
34      //阻塞线程 1
35      err = pthread_join(threadid1, &tret);
36      if (err != 0)
37      {
38         printf("阻塞线程 1 失败，错误为: %s\n", strerror(err));
39      }
40      //退出并且打印线程 1 的退出状态
41      printf("线程 1 的退出码为  %d\n", (int)tret);
42      //阻塞线程 2
43      err = pthread_join(threadid2, &tret);
44      if (err != 0)
45      {
46         printf("阻塞线程 2 失败，错误为: %s\n", strerror(err));
47      }
48      //退出并且打印线程 2 的退出状态
49      printf("线程 2 的退出码为  %d\n", (int)tret);
```

```
50        exit(0);
51    }
```

将文件保存为 exam104threadjoinquit.c，在终端使用 gcc 进行编译链接，生成可执行文件。

```
alloy@ubuntu:~/linuxc/chapter10$ gcc exam104threadjoinquit.c -o exam104threadjoinquit -lpthread
```

运行可执行文件，可以看到线程 1、2 依次退出，然后输出了它们的退出码。

```
alloy@ubuntu:~/linuxc/chapter10$ ./exam104threadjoinquit
thread 1 returning
thread 2 exiting
线程 1 的退出码为  1
线程 2 的退出码为  2
```

10.2.3　取消和清理线程

在 Linux 操作系统中，线程可以通过调用 pthread_cancel 函数来请求取消同一进程中的其他线程，对该函数的标准调用格式说明如下：

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

函数的参数是需要取消线程的线程标识符，当操作成功的时候返回“0”，否则会返回对应的错误编号。

在调用 pthread_cancel 取消了一个线程之后，需要调用相应的函数对进程退出之后的环境进行清理，这些函数被称为线程清理处理程序（Thread Cleanup Handler），线程可以建立多个清理处理程序，对这些函数的标准调用格式说明如下：

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *),void *arg);
void pthread_cleanup_pop(int execute);
```

pthread_cleanup_push 函数将子程序 routine 连同它的参数 arg 一起压入当前线程的 cleanup 处理程序的堆栈；当当前线程调用 pthread_exit 或者是通过 pthread_cancel 终止执行时，堆栈中的处理程序将按照压栈时的相反顺序依次调用。

而函数 pthread_cleanup_pop 从线程的 cleanup 处理程序堆栈中弹出最上面的一个处理程序并执行它。

pthread_cleanup_push 函数和 pthread_cleanup_pop 函数都没有返回值。

需要注意的是，其实真正对线程执行清理工作的是在 pthread_cleanup_push 中作为参数传递进去的 routine 函数，其参数通过 arg 传递进去，其在线程执行如下动作的时候被调用：

- 调用 pthread_exit 函数的时候。
- 响应取消请求的时候。
- 利用非 execute 参数调用 pthread_cleanup_pop 的时候。

如果 execute 参数被置为“0”时，清理函数将不会被调用，无论在哪种情况下，

pthread_cleanup_pop 都将删除 pthread_cleanup_push 调用建立的清理处理程序。

10.2.4　分离线程

在 Linux 中，线程一般有分离和非分离两种状态。在默认的情形下线程是非分离状态，父线程维护子线程的某些信息并等待子线程的结束，在没有显示调用 join 的情形下，子线程结束时，父线程维护的信息可能没有得到及时释放，如果父线程中大量创建非分离状态的子线程（在 Linux 系统中使用 pthread_create 函数），可能会出现堆栈空间不足的错误，其出错的返回值是 12。而对于分离线程来说，不会有其他的线程等待它的结束，运行结束后，线程终止，资源及时释放。

在 Linux 内核中，可以调用 pthread_detach 函数来进行线程的分离，对其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

其参数是需要分离的线程标识符，如果函数调用成功则返回“0”，如果调用失败则返回错误编号。

例 10.5 是一个线程分离的应用实例。

【例 10.5】线程分离

应用代码在主函数中使用 for 循环创建了 20 个线程，然后使用 pthread_detach 函数将创建出来的线程分离；这些线程都使用相同的线程入口函数 threaddeal，该函数的用途是在屏幕上输出一个字符串，用于显示这是当前的线程编号，该编号由 pthread_create 函数的 arg 参数传递给线程入口函数。

实例的应用代码如下：

```
1   #include <errno.h>
2   #include <pthread.h>
3   #include <stdlib.h>
4   #include <stdio.h>
5   #include <unistd.h>
6   //线程处理函数
7   void *threaddeal(void *arg)
8   {
9      int i = *(int *)(arg);
10     printf("这是第%d 个线程\n",i);
11  }
12  //主程序
13  int main(void)
14  {
15     //线程 id
16     pthread_t threadid;
17     int j;
18     //创建大量线程
19     int count = 20;   //多次循环
```

```
20      for(j=0 ; j < count ; j++)
21      {
22        //线程参数
23        int * p = &(j);
24        //创建线程
25        int ret= pthread_create(&threadid, NULL, threaddeal, (void*)p);
26        if(ret)//创建失败
27        {
28          printf("创建线程失败:%d\n",ret);
29        }
30        else//创建成功
31        {
32          //分离线程回收线程 stack 占用的内存
33          pthread_detach(threadid);
34        }
35      }
36      return 0;
37    }
```

将文件保存为 exam105threaddetach.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam105threaddetach。

```
alloy@ubuntu:~/linuxc/chapter10$ gcc exam105threaddetach.c -o exam105threaddetach -lpthread
```

运行该可执行文件，可以看到对应的字符串输出，需要注意的是由于这些线程彼此之间并不同步，所以会出现序号错误。

```
alloy@ubuntu:~/linuxc/chapter10$ ./exam105threaddetach
这是第 1 个线程
这是第 2 个线程
这是第 3 个线程
这是第 6 个线程
这是第 7 个线程
这是第 6 个线程
这是第 10 个线程
这是第 12 个线程
这是第 11 个线程
这是第 4 个线程
这是第 16 个线程
这是第 16 个线程
这是第 19 个线程
这是第 16 个线程
这是第 16 个线程
这是第 16 个线程
```

10.3 线程的属性

在例 10.1~10.5 中调用 pthread_create 函数来创建一个线程的时候，其传入的参数都是空指针，

而不是一个指向 pthread_attr_t 结构的指针，其实在实际应用中，可以对线程的属性进行相应的操作，对线程的属性说明如下：

```
typedef struct
{
        int         detachstate;
        int         schedpolicy;
        struct      sched_param     schedparam;
        int         inheritsched;
        int         scope;
        size_t      guardsize;
        int         stackaddr_set;
        void        *stackaddr;
        size_t      stacksize;
} pthread_attr_t;
```

对该结构中的各个参数说明如下：

- datachstate：表示线程的分离状态。
- schedpolicy：表示线程的调度策略。
- schedparam：表示线程的调度参数。
- inheritsched：表示线程的继承性。
- scope：表示线程的作用域。
- stackaddr_set：表示线程堆栈的位置，通常来说这是线程堆栈的最低位置。
- stacksize：表示线程堆栈的大小。

10.3.1 线程属性对象的初始化和销毁函数

在使用一个线程属性对象之前，必须对其进行初始化，pthread_attr_init 函数用于完成对线程属性对象初始化；在使用完一个线程属性对象后，必须对其进行销毁，pthread_attr_destroy 函数用于完成对线程属性对象的销毁，对其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

函数 pthread_attr_init 和 pthread_attr_destroy 都只有 1 个参数，此参数为一个指向线程属性对象的指针。

这两个函数在调用成功时返回 0，失败时返回-1。

10.3.2 线程堆栈大小相关函数

函数 pthread_attr_setstacksize 和 pthread_attr_getstacksize 分别用来设置和得到线程堆栈的大小，这两个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

```
int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize);
```

这两个函数具有两个参数，第 1 个是指向属性对象的指针，第 2 个是堆栈大小或指向堆栈大小的指针。

这两个函数在成功调用时返回 0，失败时返回-1。

10.3.3 线程堆栈地址函数

函数 pthread_attr_setstackaddr 和 pthread_attr_getstackaddr 分别用来设置和得到线程堆栈的位置，这两个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stack_addr);
int phread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr);
```

这两个函数具有两个参数，第 1 个是指向属性对象的指针，第 2 个是堆栈地址或指向堆栈地址的指针。

这两个函数在成功调用时返回 0，失败时返回-1。

10.3.4 线程的分离状态函数

函数 pthread_attr_setdetachstate 和 pthread_attr_getdetachstate 分别用来设置和得到线程的分离状态，这两个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int phread_attr_getdetachstate(const pthread_attr_t *attr, int* detachstate);
```

这两个函数具有两个参数，第 1 个是指向属性对象的指针，第 2 个是分离状态或指向分离状态的指针。分离状态的可能值是 PTHREAD_CREATE_JOINABLE 或是 PTHREAD_CREATE_DETACHED，缺省值是前者。

在可联合的状态中，另外一个线程可以通过 pthread_join 函数来同步线程的终止，而且可以恢复线程的终止代码，但是有一些线程的资源在线程退出后并不会释放，这样其他线程在创建时可以重新利用这些资源。

在脱离状态下，线程的资源在线程结束后立刻释放，而且不能用 pthread_join 函数来同步线程的终止。

这两个函数在成功调用时返回 0，失败时返回-1。

10.3.5 线程的作用域函数

函数 pthread_attr_setscope 和 pthread_attr_getscope 分别用来设置和得到线程的作用域，这两个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int phread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

两个函数具有两个参数，第 1 个是指向属性对象的指针，第 2 个是作用域或指向作用域的指针。作用域控制线程是否在进程内或在系统级上竞争资源，可能的值是 PTHREAD_SCOPE_PROCESS 或是 PTHREAD_SCOPE_SYSTEM。系统默认值为：PTHREAD_SCOPE_SYSTEM。

这两个函数在成功调用时返回 0，失败时返回-1。

10.3.6 线程的继承调度函数

继承调度的意思是当新创建一个线程时，线程的调度策略和调度参数是由 schedpolicy 和 schedparam 属性指定还是从创建它的父线程中继承。函数 pthread_attr_setinheritsched 和 pthread_attr_getinheritsched 分别用来设置和得到线程的继承调度，这两个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
int phread_attr_getinheritsched (const pthread_attr_t *attr, int *inherit);
```

这两个函数具有两个参数，第 1 个是指向属性对象的指针，第 2 个是继承调度或指向继承调度的指针。继承调度的可能值是 PTHREAD_EXPLICIT_SCHED 或是 PTHREAD_INHERIT_SCHED，分别对应上面两种情况，系统的默认值为 PTHREAD_EXPLICIT_SCHED。

这两个函数在成功调用时返回 0，失败时返回-1。

10.3.7 线程的调度策略函数

函数 pthread_attr_setschedpolicy 和 pthread_attr_getschedpolicy 分别用来设置和得到线程的调度策略，这两个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy (const pthread_attr_t *attr, int * policy);
```

这两个函数具有两个参数，第 1 个是指向属性对象的指针，第 2 个是调度策略或指向调度策略的指针。调度策略可能的值是先进先出（SCHED_FIFO）、轮转法 （SCHED_RR），或是其他未定义（SCHED_OTHER）。调度策略的默认值是 SCHED_OTHER。

调度策略 SCHED_RR 和 SCHED_FIFO 仅仅对有超级用户权限的进程才有效。

这两个函数在成功调用时返回 0，失败时返回-1。

10.3.8 线程的调度参数函数

函数 pthread_attr_setschedparam 和 pthread_attr_getschedparam 分别用来设置和得到线程的调度参数，这两个函数的原型如下所示：

```
#include <pthread.h>
int pthread_attr_setschedparam (pthread_attr_t *attr,const struct sched_param *param);
int pthread_attr_getschedparam (const pthread_attr_t *attr, struct sched_param *param);
```

这两个函数具有两个参数，第 1 个是指向属性对象的指针，第 2 个参数是 sched_param 结构或指向该结构的指针。结构 sched_param 在文件/usr/include/bits/sched.h 中定义，如下所示：

```
struct sched_param
{
        int sched_priority;
};
```

结构 sched_param 的子成员 sched_priority 控制一个优先权值，大的优先权值对应高的优先权。系统默认的调度参数是：优先级 0。

如果线程的调度策略是 SCHED_OTHER，那么这个参数就可以忽略。只有当线程的调度策略 SCHED_RR 或者 SCHED_FIFO 时，这个参数才有用。

这两个函数在成功调用时返回 0，失败时返回-1。

10.3.9 使用线程的属性

例 10.6 是一个使用线程属性来传递线程分离状态的实例，其没有使用线程分离函数 pthread_detach，而是通过修改线程的属性以实现线程分离。

【例 10.6】使用线程的属性

应用代码定义了线程的属性对象 thread_attr，然后调用 pthread_attr_init 函数对该属性对象进行初始化，使用 pthread_attr_setdetachstate 将属性对象设置为分离状态，并且在创建新线程的时候将该属性对象传递给新线程；threaddeal 函数是新线程的入口处理函数，其首先输出一个字符串表明当前正在执行线程，然后调用 sleep 函数自我休眠 3 秒之后再次输出字符串表明即将退出，并且将一个标志位 thread_flag 的状态修改为 FALSE；主程序则在创建了新线程之后等待该标志位状态改变，然后退出。

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <pthread.h>
4   #define TRUE 1                                  //定义两个常量
5   #define FALSE 0
6   int thread_flag = TRUE;                         //标志位
7   //线程处理函数
8   void *threaddeal(void *arg)
9   {
10     printf("当前线程正在执行.\n");
11     sleep(3);                                    //休眠 3 秒
12     printf("线程即将退出.\n");
13     thread_flag = FALSE;                         //修改线程标志位
14     pthread_exit(NULL);                          //线程退出
15  }
16  //主程序
17  int main(int argc,char *argv[])
```

```
18  {
19      pthread_t threadid;                          //定义线程描述符
20      pthread_attr_t thread_attr;                  //定义线程属性对像
21      pthread_attr_init(&thread_attr);             //线程属性初始化
22      pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);
        //将线程设置为分离状态
23      if(pthread_create(&threadid, &thread_attr, threaddeal, NULL))
        //创建新线程，并修改属性
24      {
25          printf("创建线程失败\n");
26          exit(0);
27      }
28      while(thread_flag)      //判断标志位
29      {
30          printf("等待线程结束\n");
31          sleep(1);
32      }
33      printf("线程结束.\n");
34      return 0;
35  }
```

将文件保存为 exam106setattr.c，然后在终端中使用 gcc 进行编译链接，生成可执行文件。

```
alloy@ubuntu:~/linuxc/chapter10$ gcc exam106setattr.c -o exam106setattr -lpthread
```

执行该可执行文件，可以看到主程序一直在等待线程执行结束后修改标志位，由于线程调用了 sleep 函数用于休眠 3 秒，而主程序只休眠 1 秒，所以会看到主程序三次输出“等待线程结束”。

```
alloy@ubuntu:~/linuxc/chapter10$ ./exam106setattr
等待线程结束
当前线程正在执行.
等待线程结束
等待线程结束
线程即将退出.
线程结束.
```

10.4 线程的私有数据

每个线程都有一些属于自己的数据，当线程对这些数据进行操作的时候可以独立地访问它们，而不用担心其他线程和自己争夺所有权，这种数据被称为线程私有数据（线程特定数据），其是存储和查询与某个线程相关的数据的一种机制。

引入线程的私有数据机制是为了解决两个问题：

- 有些时候是需要维护每个线程的数据，因为线程标识符并不是一个小而连续的整数，所以不能简单地分配一个线程数据数组。
- 线程私有数据提供了让基于进程的接口适应多线程环境的机制。

Linux 内核提供了很多对线程私有数据的操作函数。

10.4.1　创建键函数

在分配线程私有数据之前，需要创建和该数据相关联的键，这个键用于获取对线程私有数据的访问权，用户可以使用 pthread_key_create 函数来创建一个键，对其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *key, void(*dest_routine(void *)));
```

函数 pthread_key_create 用于创建一个对进程中的所有线程都可见的关键字，该关键字可以通过函数 pthread_setspecific 和 pthread_getspecific 来读取和设置。

当创建一个关键字时，进程中所有线程的这个关键字值都为 NULL，当创建一个线程时，这个线程的所有关键字的值都为 NULL。

如果 pthread_key_create 执行成功，则返回 0，并在参数 key 中保存新创建的关键字 ID，如果调用失败则返回其他值。

除了创建键以外，pthread_key_create 可以选择为该键关联一个析构函数，当线程退出的时候，如果数据地址已经被置为一个非 NULL 的值，则这个析构函数会被调用，该函数的唯一参数就是该数据地址。

线程可以为线程私有数据分配多个键，每个键都可以由一个析构函数与其关联，各个键的析构函数可以互不相同。

10.4.2　取消键关联函数

对于用户而言，可以调用 pthread_key_delete 来取消键与线程私有数据值之间的关联关系，对其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t key);
```

其参数 key 为需要取消键的标号，如果调用成功则返回“0”，如果调用失败则返回错误编号。

需要注意的是这个函数在调用的时候并不会激活与键关联的析构函数，需要释放任何与键对应的线程私有数据值的内存空间。

10.4.3　解决键冲突函数

有些线程可能看到某个键值，而其他的线程看到的则是另外一个不同的值，这是一种竞争，如果需要解决这种竞争可以使用 pthread_once 函数。

```
#include <pthread.h>
void * pthread_once_t once_control=PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

pthread_once 函数用于保证某些初始化代码至多只能执行一次，参数 once_control 指向静态的或外部的变量，这个变量初始化为 PTHREAD_ONCE_INIT。

当第一次调用 pthread_once 时，系统将记录已经执行了初始化，后面再调用 pthread_once 时，如果参数 once_control 相同，那么什么也不做，该函数的返回值一定是“0”。

10.4.4　键关联函数

当键被创建之后，可以调用 pthread_setspecific 函数，对其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void *pointer);
```

函数 pthread_setspecific 指定由参数 pointer 指定的指针指向由参数 key 指定的关键字。每一个线程都有一个互相独立的指针，这个指针指向一个特定的关键字。

10.4.5　线程私有数据地址获取函数

可以通过 pthread_getspecific 函数来获取线程私有数据的地址，对其标准调用格式说明如下：

```
#include <pthread.h>
void * pthread_getspecific(pthread_key_t key);
```

函数 pthread_getspecific 用来获取由 pthread_setspecific 设置的关键字指针，如果调用成功，则返回一个指向最近一次使用 pthread_setspecific 而设定的指向线程关键字的指针。

如果没有线程私有数据和键关联，此函数将返回一个空指针，可以根据其返回值来确定是否要调用键关联函数。

10.4.6　使用线程的私有数据

例 10.7 给出了一个使用线程的私有数据实例，这是一个使用静态变量来累加调用结果的库函数应用，这种累加表现为将线程返回的字符串连接起来。

【例 10.7】使用线程的私有数据

应用代码首先预定了下列函数用于进行对应的操作。

- char * str_accumulate(const char *s)：字符串处理函数，用于将参数 s 传递的字符串和线程的私有数据对应的字符串连接到一起。
- static void str_alloc_key()：这是一个创建线程私有数据的函数。
- static void str_alloc_destroy_accu(void *accu)：这是用于撤销线程私有数据的函数。
- void *threaddeal(void *arg)：这是线程的入口处理函数，其主要工作是调用 str_accumulate 函数将 arg 存放的字符串和“Result of *thread”连接到一起。

在主程序中创建了两个线程，然后阻塞它们，等待对应的输出。

实例的应用代码如下：

```
1   #include <stddef.h>
2   #include <stdio.h>
3   #include <stdlib.h>
```

```
 4  #include <string.h>
 5  #include <pthread.h>
 6  #if 0      //预定义
 7  char * str_accumulate(char *s)
 8  {
 9          static char accu[1024]={0};
10          strcat(accu,s);
11          return accu;
12  }
13  #endif
14      static pthread_key_t str_key;                                        //一个键值
15      static pthread_once_t str_alloc_key_once = PTHREAD_ONCE_INIT;        //用于解决键冲突
16      static void str_alloc_key();                                         //按键分配函数
17      static void str_alloc_destroy_accu(void *accu);                      //撤销按键分配函数
18      //处理函数
19      char * str_accumulate(const char *s)
20      {
21          char *accu;
22          pthread_once(&str_alloc_key_once,str_alloc_key);            //解决按键冲突
23          accu = (char *)pthread_getspecific(str_key);                //获取线程的私有数据地址
24          if(accu == NULL)
25          {
26              accu = malloc(1024);      //分配 1024 的空间
27              if(accu == NULL)   //如果 accu 为 NULL 则直接返回 NULL
28              {
29                  return NULL;
30              }
31              accu[0] = 0;
32              pthread_setspecific(str_key,(void *)accu);          //将 accu 存放的数据作为键值关联
33              printf("Thread %lx : allocating buffer at %p\n",pthread_self(),accu);          //打印输出
34          }
35          strcat (accu,s);                                        //将 accu 和 s 字符串连接到一起
36          return accu;
37      }
38      //这是一个键值分派函数
39      static void str_alloc_key()
40      {
41          pthread_key_create(&str_key,str_alloc_destroy_accu);                 //创建键值
42          printf("Thread %lx : allocated key %d\n",pthread_self(), str_key);
43      }
44      //这是撤销键值的函数
45      static void str_alloc_destroy_accu(void *accu)
46      {
47          printf("Thread %lx : freeing buffer at %p\n",pthread_self(),accu);
48          free(accu);                                                          //释放空间
49      }
50      //线程处理函数
51      void *threaddeal(void *arg)
52      {
```

```
53          //该函数的主要工作是将 arg 的字符串和 "Result of *thread 连接到一起"
54          char *str;
55          str = str_accumulate("Result of ");
56          str = str_accumulate((char *)arg);
57          str = str_accumulate(" thread");
58          printf("Thread %lx: \"%s\" \n",pthread_self(),str);
59          return NULL;
60      }
61  //主函数
62  int main(int argc, char *argv[])
63  {
64      char *str;
65      pthread_t th1,th2;
66      str = str_accumulate("Result of ");
67      pthread_create(&th1,NULL,threaddeal,(void *)"first");
68      pthread_create(&th2,NULL,threaddeal,(void *)"second");      //建立两个线程
69      str = str_accumulate("initial thread");
70      printf("Thread %lx :\"%s\"\n",pthread_self(),str);
71      pthread_join(th1,NULL);
72      pthread_join(th2,NULL);                                   //阻塞线程 1 和线程 2
73      return 0;
74  }
```

将文件保存为 exam107pthreadkey.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam107pthreadkey。

```
alloy@ubuntu:~/linuxc/chapter10$ gcc exam107pthreadkey.c -o exam107pthreadkey -lpthread
```

执行该可执行文件，可以看到对应的字符串输出。

```
alloy@ubuntu:~/linuxc/chapter10$ ./exam107pthreadkey
Thread 7f3a7ec7f700 : allocated key 0
Thread 7f3a7ec7f700 : allocating buffer at 0xec6010
Thread 7f3a7ec7f700 :"Result of initial thread"
Thread 7f3a7e49d700 : allocating buffer at 0x7f3a780008c0
Thread 7f3a7e49d700: "Result of first thread"
Thread 7f3a7e49d700 : freeing buffer at 0x7f3a780008c0
Thread 7f3a7dc9c700 : allocating buffer at 0x7f3a700008c0
Thread 7f3a7dc9c700: "Result of second thread"
Thread 7f3a7dc9c700 : freeing buffer at 0x7f3a700008c0
```

10.5 线程的同步

和进程类似，线程也存在同步的问题，当多个控制线程共享相同的内存时，需要确保每个线程看到一致的数据视图，如果每个线程使用的变量都是其他线程不会读取或者修改的，就不会存在一致性问题，否则就需要注意同步问题。通常来说用户可以使用互斥量或者条件变量方式来解决线程的同步问题。

10.5.1 使用互斥锁解决线程同步

互斥锁是一个简单的锁定命令，它可以用来锁定对共享资源的访问。对于线程来说，整个地址空间都是共享的资源，所以线程的任何资源都是共享的；互斥锁具有以下三个主要特点。

- 原子性：把一个互斥锁定为一个原子操作，这意味着操作系统（或 pthread 函数库）保证了如果一个线程锁定了一个互斥锁，则没有其他线程可以在同一时间成功锁定这个互斥锁。
- 唯一性：如果一个线程锁定一个互斥锁，在它解除锁定之前，没有其他线程可以锁定这个互斥量。
- 非繁忙等待：如果一个线程已经锁定了一个互斥锁，第二个线程又试图去锁定这个互斥锁，则第二个线程将被挂起（不占用任何 CPU 资源），直到第一个线程解除对这个互斥锁的锁定为止，第二个线程则被唤醒并继续执行，同时锁定这个互斥锁。

Linux 内核提供了相应的函数来完成对应的操作。

1. 互斥锁的初始化函数

pthread_mutex_init 用来初始化一个由参数 mutex 指向的互斥锁，这个互斥锁的属性由参数 attr 指定，或者通过指定 attr 为 NULL 而使用默认的属性，对其标准调用格式说明如下：

```
#include <pthread.h>
pthread_mutex_t fastmutex=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex=PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t errchkmutex=PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutex_attr *attr);
```

上面三个常量是常用的处理互斥锁的常量。

不会出现有多个线程同时初始化同一个互斥锁的情形，一个互斥锁在使用期间一定不会被重新初始化。

如果 pthread_mutex_init 执行成功，则返回 0，并将新创建的互斥锁的 ID 值放到参数 mutex 中。如果执行失败，那么将返回一个错误编号。

2. 互斥锁解除函数

pthread_mutex_destroy 函数用于解除由参数 mutex 指向的互斥锁的任何状态，对其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

需要注意的是存储互斥锁的内存并不被释放，如果 pthread_mutex_destroy 执行成功，则返回 0；如果执行失败，那么将返回一个错误编号。

3. 互斥锁锁定函数

pthread_mutex_lock 函数可以用于锁定由参数 mutex 指向的互斥锁，对其标准调用格式说明如

下：

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

如果 mutex 已经被锁定，那么当前调用的线程将阻塞直到互斥锁被其他线程释放（阻塞线程按照线程优先级等待)。当 pthread_mutex_lock 返回时，说明互斥锁已经被当前线程成功加锁。

如果 pthread_mutex_lock 执行成功则返回 0，其他的值说明发生了错误。

4. 互斥锁加锁函数

pthread_mutex_trylock 函数用于尝试给由参数 mutex 指定的互斥锁加锁，对其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

该函数是 pthread_mutex_lock 的非阻塞版本。pthread_mutex_lock 在给一个互斥锁加锁时，如果互斥锁已经被锁定，那么 pthread_mutex_lock 将一直阻塞，不会立即返回。而使用 pthread_mutex_trylock 给一个互斥锁加锁时，如果互斥锁已经被锁定，那么 pthread_mutex_trylock 调用将返回错误，否则，互斥锁将被调用者加锁。

如果 pthread_mutex_trylock 执行成功则返回 0，其他值意味着错误。

5. 互斥锁解锁函数

可以使用 pthread_mutex_unlock 函数给由参数 mutex 指定的互斥锁解锁，对其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

互斥锁必须处于加锁状态，而且调用本函数的线程必须是给互斥锁加锁的同一个线程才能给互斥锁解锁。如果有其他线程在等待互斥锁，那么由核心的调度程序决定哪个线程将获得互斥锁并脱离阻塞状态。

如果 pthread_mutex_unlock 执行成功，则返回 0，其他值意味着错误。

6. 使用互斥锁

如下是一个使用互斥锁完成线程中一个公共变量的操作，并且输出其当前数值的实例。

【例 10.8】使用互斥锁完成线程同步

应用代码创建了两个线程，同时对全局变量 x 进行减 1 操作，并且输出当前的 x 数值，线程 1 和线程 2 对应的入口处理函数分别为 threaddeal1 和 threaddeal2，其操作内容都是相同的，首先调用 pthread_mutex_lock 函数对互斥量进行加锁操作，然后输出提示字符串，将全局变量进行减 1 操作，随后调用 pthread_mutex_unlock 函数去掉互斥量的锁，休眠 1 秒后退出，直到全局变量的 x 等于 0 为止；主程序将 x 初始化为 10，创建两个线程后阻塞等待线程，执行完成后退出。

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <pthread.h>
4
5   pthread_mutex_t mutex;                          //定义一个互斥量
6   int x;                                          //定义一个全局变量
7   //这是线程 1 的入口函数
8   void threaddeal1(void)
9   {
10      while(x>0)                                  //如果 X>0
11      {
12          pthread_mutex_lock(&mutex);             //对互斥量进行加锁操作
13          printf("线程 1 正在运行: x=%d \n",x);    //输出当前的 x 值
14          x--;                                    //将 x 的值-1
15          pthread_mutex_unlock(&mutex);           //对互斥量进行开锁操作
16          sleep(1);                               //休眠 1 秒
17      }
18      pthread_exit(NULL);                         //进程退出
19  }
20  //这是线程 2 的入口函数，线程 2 和线程 1 的操作完全相同
21  void threaddeal2(void)
22  {
23      while(x>0)
24      {
25          pthread_mutex_lock(&mutex);
26          printf("线程 2 正在运行: x=%d \n",x);
27          x--;
28          pthread_mutex_unlock(&mutex);
29          sleep(1);
30      }
31      pthread_exit(NULL);
32  }
33  //这是主函数
34  int main(int argc,char *argv[])
35  {
36      pthread_t threadid1,threadid2;
37      int ret;
38      ret = pthread_mutex_init(&mutex,NULL);      //初始化互斥锁
39      if(ret != 0)
40      {
41          printf ("初始化互斥锁失败.\n");
42          exit (1);
43      }
44      x = 10;                                     //给全局变量赋初始化值
45      ret = pthread_create(&threadid1, NULL, (void *)&threaddeal1, NULL);          //创建线程 1
46      if(ret != 0)
47      {
```

```
48        printf ("创建线程 1 失败.\n");
49        exit (1);
50    }
51    ret = pthread_create(&threadid2, NULL, (void *)&threaddeal2, NULL);   //创建线程 2
52    if(ret != 0)
53    {
54        printf ("创建线程 2 失败.\n");
55        exit (1);
56    }
57    pthread_join(threadid1, NULL);
58    pthread_join(threadid2, NULL);                    //阻塞线程 1 和线程 2
59    return (0);
60  }
```

将文件保存为 exam108pthreadmutex.c，在终端中使用 gcc 进行编译链接，生成可执行文件。

```
alloy@ubuntu:~/linuxc/chapter10$ gcc exam108pthreadmutex.c -o exam108pthreadmutex -lpthread
```

执行该可执行文件，可以看到线程轮流将公共变量 x 进行减 1 操作，然后输出当前的 x 值。

```
alloy@ubuntu:~/linuxc/chapter10$ ./exam108pthreadmutex
线程 1 正在运行: x=10
线程 2 正在运行: x=9
线程 1 正在运行: x=8
线程 2 正在运行: x=7
线程 1 正在运行: x=6
线程 2 正在运行: x=5
线程 1 正在运行: x=4
线程 2 正在运行: x=3
线程 1 正在运行: x=2
线程 2 正在运行: x=1
```

10.5.2 使用条件变量解决线程同步

在程序中使用互斥锁虽然可以解决一些资源竞争的问题，但是互斥锁只有两种状态，这使得它的用途非常有限。

除了互斥锁之外，还可以使用条件变量来解决线程的同步问题，条件变量是对互斥锁的补充，它允许线程阻塞并等待另一个线程发送的信号。当收到信号时，阻塞的线程就被唤醒并试图锁定与之相关的互斥锁。

Linux 同样提供了相应的函数来完成对应的操作。

1. 条件变量初始化函数

pthread_cond_init 函数用于初始化由参数 cond 指定的条件变量，对其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond, const pthread_cond_attr *attr);
```

这个条件变量的属性由参数 attr 指定。如果参数 attr 为 NULL，那么就使用默认的属性设置。

多线程不能同时初始化同一个条件变量，如果一个条件变量正在使用，则它不能被重新初始化。

如果 pthread_cond_init 执行成功，则返回 0，并将新创建的条件变量的 ID 放在参数 cond 中，如果返回其他的值则意味着有错误。

2. 条件变量解除函数

pthread_cond_destroy 函数用于清除由参数 cond 指向的条件变量的任何状态，对其标准调用格式说明如下：

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

需要注意的是存储条件变量的内存空间不被释放，如果函数 pthread_cond_destroy 执行成功则返回 0，其他值意味着错误。

3. 条件变量阻塞函数

函数原型：

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

使用 pthread_cond_wait 释放由参数 mutex 指向的互斥锁，被阻塞的线程可以被 pthread_cond_signa1、pthread_cond_broadcast 或者由 fork 和传递信号引起的中断唤醒。

即使返回错误信息，pthread_cond_wait 通常在互斥锁被调用线程加锁后才返回。函数将阻塞直到条件变量被信号唤醒，它在阻塞前自动释放互斥锁，在返回前再自动获得它。如果有多个线程关于条件变量阻塞，则其退出阻塞状态的顺序将不确定。如果 pthread_cond_wait 执行成功则返回 0，其他值意味着错误。

4. 带时间的条件变量阻塞函数

函数原型：

```
#include <pthread.h>
int pthread_cond_timewait(pthread_cond_t *cond, pthread_mutex_t *mutex,
const struct timespec *abstime);
```

pthread_cond_timewait 和 pthread_cond_wait 的用法相似，区别在于 pthread_cond_timewait 在经过由参数 abstime 指定的时间时不阻塞。

即使是返回错误，pthread_cond_timewait 也只在给互斥锁加锁后返回。

pthread_cond_timewait 函数将阻塞，直到条件变量获得信号或者经过由 abstime 指定的时间。

如果 pthread_cond_timewait 执行成功则返回零。如果阻塞条件变量的时间超过了由参数 abstime 所指定的时间，那么就返回 ETIMEOUT，其他值意味着错误。

5. 单个条件变量阻塞退出函数

函数原型：

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

使用 pthread_cond_signal 使得由参数 cond 指向的条件变量阻塞的线程退出阻塞状态。在同一个互斥锁的保护下使用 pthread_cond_signal，否则，条件变量可以在对关联条件变量的测试和 pthread_cond_wait 带来的阻塞之间获得信号，这将导致无限期的等待。

如果没有一个线程关于条件变量阻塞，那么 pthread_cond_signal 无效。

如果 pthread_cond_signal 执行成功则返回 0，其他值意味着错误。

6. 全部条件变量阻塞退出函数

函数原型：

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
```

使用 pthread_cond_broadcast 使得所有由参数 cond 指向的条件变量阻塞的线程退出阻塞状态。如果没有阻塞的线程，则 cond_broadcast 无效。

这个函数将唤醒所有由 pthread_cond_wait 阻塞的线程，因为所有关于条件变量阻塞的线程都同时参与竞争，所以使用这个函数时需要小心。

如果 pthread_cond_broadcast 执行成功则返回 0，其他值意味着错误。

7. 使用条件变量

在 Linux 中有一个经典的同时性编程问题，即“生产者-消费者”问题，该问题描述的是存在一个有限缓冲区和两个线程：生产者和消费者，前者分别不停地把产品放入缓冲区，而后者从缓冲区中拿走产品；生产者在缓冲区满的时候必须等待，消费者在缓冲区空的时候也必须等待。此外因为缓冲区是临界资源，所以生产者和消费者之间必须互斥执行，它们之间的关系如图 10.2 所示，其本质即为两个线程的同步操作。

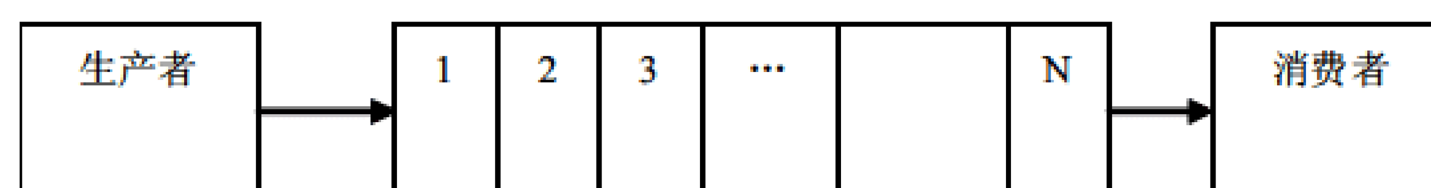


图 10.2 “生产者-消费者”问题模型

例 10.9 是一个使用条件变量来解决“生产者-消费者”问题的实例。

【例 10.9】使用条件变量完成线程同步

应用代码首先定义了如下一个结构体作为生产者（producers）的条件结构变量，其中对各个分量的说明可参考注释部分：

```
struct producers                                  //定义生产者条件变量结构
{
        int buffer[BUFFER_SIZE];                  //定义缓冲区
        pthread_mutex_t lock;                     //定义访问缓冲区的互斥锁
        int             readpos, writepos;        //读写的位置
        pthread_cond_t   notempty;                //缓冲区有数据时的标记
```

```
            pthread_cond_t    notfull;                          //缓冲区未满的标记
    };
```

然后定义了两个函数 put 和 get 分别用于向缓冲区中放入一个数据以及从缓冲区中读出一个数据，还分别使用 producer 和 consumer 作为生产者和消费者线程的入口处理函数。在预定义中将缓冲区大小设置为 4，将缓冲区溢出状态 OVER 预定义为-1。在主程序中首先初始化缓冲区，然后创建两个线程，分别对应生产者和消费者，阻塞这两个线程等待退出。

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <pthread.h>
3   #define BUFFER_SIZE 4
4   #define OVER (-1)
5   struct producers                                    //定义生产者条件变量结构
6   {
7           int buffer[BUFFER_SIZE];                    //定义缓冲区
8           pthread_mutex_t lock;                       //定义访问缓冲区的互斥锁
9           int             readpos, writepos;          //读写的位置
10          pthread_cond_t  notempty;                   //缓冲区有数据时的标记
11          pthread_cond_t  notfull;                    //缓冲区未满的标记
12  };
13  //初始化缓冲区
14  void init(struct producers *b)
15  {
16          pthread_mutex_init(&b->lock,NULL);
17          pthread_cond_init(&b->notempty,NULL);
18          pthread_cond_init(&b->notfull,NULL);
19          b->readpos=0;
20          b->writepos=0;
21  }
22  //在缓冲区中存放一个整数
23  void put(struct producers *b, int data)
24  {
25          pthread_mutex_lock(&b->lock);
26          //当缓冲区为满时等待
27          while((b->writepos+1)%BUFFER_SIZE == b->readpos)
28          {
29            pthread_cond_wait(&b->notfull,&b->lock);
30            //在返回之前，pthread_cond_wait 需要参数 b->lock
31          }
32          //向缓冲区中写数据，并将写指针向前移动
33          b->buffer[b->writepos] = data;
34          b->writepos++;
35          if(b->writepos >= BUFFER_SIZE)
36          {
37           b->writepos=0;
38          }
39          //发送当前缓冲区中有数据的信号
```

```
40              pthread_cond_signal(&b->notempty);
41              pthread_mutex_unlock(&b->lock);
42  }
43  //从缓冲区中读数据并将数据从缓冲区中移走
44  int get(struct producers *b)
45  {
46              int data;
47              pthread_mutex_lock(&b->lock);
48              //当缓冲区中有数据时等待
49              while(b->writepos == b->readpos)
50              {
51                  pthread_cond_wait(&b->notempty,&b->lock);
52              }
53             //从缓冲区中读数据，并将指针前移
54             data = b->buffer[b->readpos];
55             b->readpos++;
56             if(b->readpos >= BUFFER_SIZE)
57             {
58                b->readpos = 0;
59             }
60             //发送当前缓冲区未满的信号
61             pthread_cond_signal(&b->notfull);
62             pthread_mutex_unlock(&b->lock);
63             return data;
64  }
65  struct producers    buffer;
66  //这是生产者的线程处理函数
67  void *producer(void *data)
68  {
69      int n;
70      for(n=0;n<10;n++)
71      {
72         printf("生产者: %d-->\n",n);                //连续 10 次生产
73         put(&buffer,n);
74      }
75      put(&buffer,OVER);                            //将状态放入 buffer 中
76      return NULL;
77  }
78  //这是消费者的线程处理函数
79  void *consumer(void *data)
80  {
81      int d;
82      while(1)
83      {
84         d = get(&buffer);                          //从 buffer 中读取对应的状态
85         if(d == OVER)                              //如果已经没有了，则停止
86         {
87           break;
88         }
```

```
89            printf("消费者: --> %d\n",d);
90        }
91        return NULL;
92    }
93    //这是主程序
94    int main(int argc,char *argv[])
95    {
96        pthread_t thproducer,thconsumer;                    //生产者和消费者的 id
97        void *retval;
98        init(&buffer);        //初始化缓冲区
99        pthread_create(&thproducer,NULL,producer,0);
100       pthread_create(&thconsumer,NULL,consumer,0);       //创建两个线程
101       pthread_join(thproducer,&retval);
102       pthread_join(thconsumer,&retval);                      //阻塞进程
103       return 0;
104   }
```

将文件保存为 exam109pthreadcont.c，在终端中使用 gcc 进行编译链接，生成可执行文件。

```
alloy@ubuntu:~/linuxc/chapter10$ gcc exam109pthreadcont.c -o exam109pthreadcont -lpthread
```

执行该可执行文件，可以看到生产者和消费者轮流生产和消费一个整数。

```
alloy@ubuntu:~/linuxc/chapter10$ ./exam109pthreadcont
生产者: 0-->
生产者: 1-->
生产者: 2-->
生产者: 3-->
生产者: 4-->
消费者: --> 0
消费者: --> 1
消费者: --> 2
消费者: --> 3
消费者: --> 4
生产者: 5-->
生产者: 6-->
生产者: 7-->
生产者: 8-->
生产者: 9-->
消费者: --> 5
消费者: --> 6
消费者: --> 7
消费者: --> 8
消费者: --> 9
```

10.6　本章习题

1. 编写一个程序，调用 pthread_create 函数创建两个线程，一个打印自己的线程 ID 和“Hello”，另一个打印自己的线程 ID 号和“Ubuntu！”

2. 使用 pthread_join 函数重写上一题的程序，使得两个线程重复打印 10 次，主进程等待两个线程都打印完成之后才退出。

3. 编写一个程序，使用 pthread_create 函数循环创建 5 个线程，然后每次在创建线程时将当前循环计数器的值通过 pthread_create 函数的 arg 参数传递给新线程，在线程中打印输出该计数器的值。

4. 编写一个程序，创建 0~4 共 5 个线程，然后每个线程输出一个 hello。

5. 编写一个程序，实现一个线程从共享的缓冲区中读数据，另一个线程向共享的缓冲区中写数据，对共享缓冲区的访问控制是通过使用一个互斥锁来实现的。

第 11 章　Linux 的网络编程

网络是 Linux 系统和外部进行数据交互的重要通道，本章将介绍在 Linux 下使用 C 语言进行网络相关编程的基础方法，涉及以下内容：

- Linux 的网络通信模型。
- 套接字基础和使用方法。
- 在 Linux 下进行 TCP 编程的方法。
- 在 Linux 进行 UDP 编程的方法。

11.1　Linux 的网络通信模型

Linux 系统和网络是息息相关的，其内核稳定支持包括 TCP/IP（IPv6）、IPX、DDP 等在内的多种网络协议，同时 Shell 也提供了多个功能强大的联网命令，例如 ftp、telnet 等。

11.1.1　OSI 网络模型

计算机网络模型是为了简化网络的研究、设计与实现而抽象出来的一种结构模型，通常采用层次模型。在每个层次模型中，往往将系统所要实现的复杂功能分化为若干个相对简单的细小功能，每一项分功能以相对独立的方式去实现。

开放系统互联参考模型 OSI（Open System Interconnection Reference Mode）是国际标准化组织（ISO）提出的一个设计和描述网络通信的基本框架，其结构如图 11.1 所示，包括了物理层、数据链路层、网络层、传输层、会话层、表示层、应用层（共 7 层），对各层的详细说明如下。



图 11.1　OSI 的网络结构模型

- 物理层（Physical Layer）：这是计算机网络的最底层，也是最基础的层，是有关物理设备通过物理媒体进行互连的描述和规定。物理层协议定义了接口的机械特性、电气特性、功能特性、规程特性，其以比特流的方式传送来自数据链路层的数据，而不去理会数据

的含义或格式。

- 数据链路层（Data Link Layer）：该层承担了两个数据设备（计算机等）通过物理层进行无差错传输数据帧的工作，通常来说这些数据帧的传输都需要等待接收方的确认，若有错误或者丢失的数据帧必须重新传送。
- 网络层（Network Layer）：该层负责信息寻址，以及将逻辑地址与名字转换为物理地址。在网络层中传输的是数据包，其需要选择合适的路径和转发数据包，使发送方的数据包能够正确无误地按地址寻找到接收方的路径，并将数据包交给接收方。网络层通常还需要对数据包进行重组以满足数据链路层对数据帧大小的要求，并且还需要考虑不同协议之间的互联问题。
- 传输层（Transport Layer）：该层负责在不同子网中的两个数据设备之间，数据包可以可靠、顺序、无错地传输，在该层中传输的是数据段，其向高层用户提供端到端的可靠的透明传输服务，为不同进程间的数据交换提供可靠的传送手段。
- 会话层（Session Layer）：其是利用传输层提供的端到端服务，向表示层或会话用户提供会话服务。会话层的主要功能是在两个节点间建立、维护和释放面向用户的连接，并对会话进行管理和控制，保证会话数据可靠传送。
- 表示层（Presentation Layer）：其负责在不同的数据格式之间进行转换操作，以实现不同计算机系统间的信息交换，以及负责编码、加密、压缩等操作。
- 应用层（Application Layer）：其直接和用户以及应用程序进行数据交互，包括了大量的应用协议，如 Telnet、SSH、DNS、HTTP 等。

通常来说可以把 OSI 网络模型的低四层（物理层、数据链路层、网络层、传输层）称为数据流层，而把高三层（会话层、表示层、应用层）称为应用层。

在 OSI 网络模型的基础上发展出了许多种类的实际应用模型，第 11.1.2 小节中即将介绍的 TCP/IP 模型即为其中一种。

11.1.2　TCP/IP 协议和其网络模型

TCP/IP（Transmission Control Protocol / Internet Protocol）是由美国国防部创建的模型，是发展至今最成功的通信协议，被应用于架构互联网（Internet），Linux 系统的网络功能也是基于该协议实现的。

1. TCP/IP 协议的分层

TCP/IP 协议是一组在网络中提供可靠数据传输和无连接数据服务的协议，其中提供可靠数据传输的协议称为传输控制协议（TCP），而提供无连接数据包服务的协议称为网际协议（IP）。



注　意

TCP/IP 协议并不是只有 TCP 和 IP 两个协议，而是包含很多其他协议的一个网络协议集合。

TCP/IP 协议的出现时间比 OSI 更早，其也有自己的网络模型，但是其并不存在和 OSI 的 7 层

严格的一一对应关系，主要是并不存在物理层和数据链路层，可以分为网络接口层、网络层、传输层和应用层共 4 个部分，图 11.2 给出了其和 OSI 模型的对应比较关系。



图 11.2　TCP/IP 协议模型和 OSI 模型的比较

TCP/IP 协议是一系列（到目前为止有 100 多个）协议的集合，这些协议分别对应 TCP/IP 协议的每个层次，如图 11.3 所示，对这些层次和协议的说明如下。

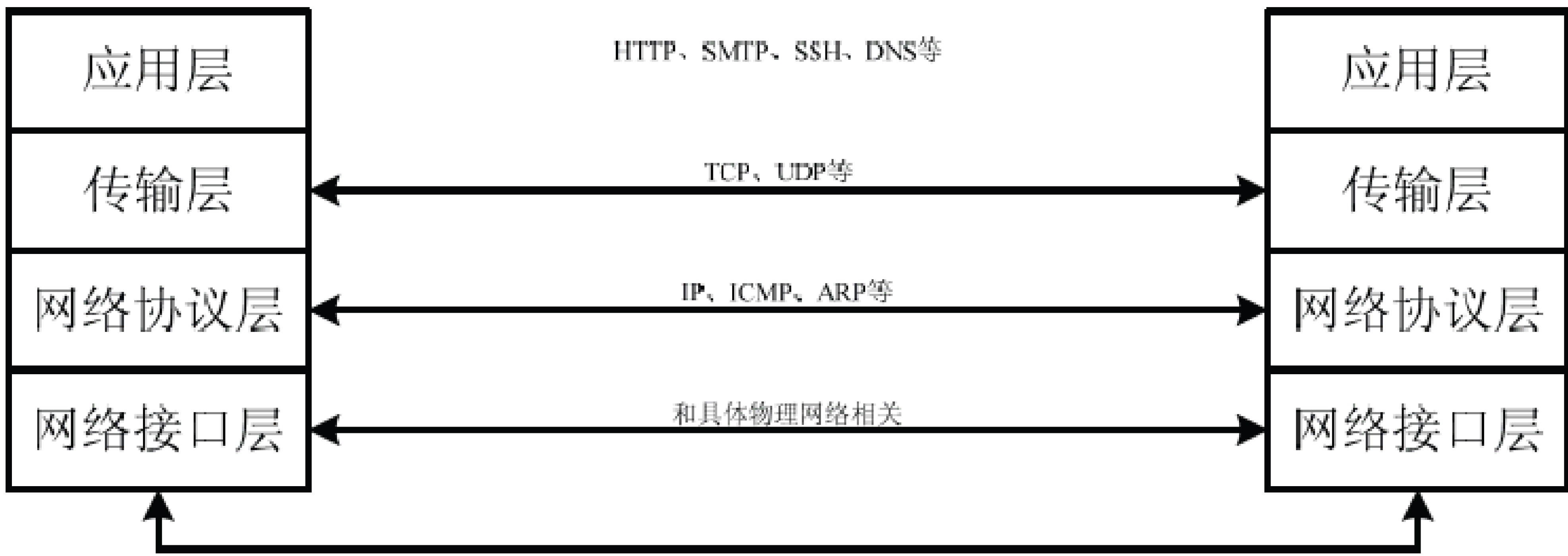


图 11.3　TCP/IP 的模型层次和对应的协议

- 应用层：其提供各种常用的高层协议，包括 FTP（文件传输）、TLENET（远程登录）、SMTP（简单邮件传送）、HTTP（超文本传输）、DNS（域名服务）等。
- 传输层：其提供了从发送方端口到接收方端口的数据传输协议，最常用的协议是 TCP 和 UDP 协议，前者是一个面向连接的协议，提供无差错的字节流的可靠传输；而后者是一个不面对连接的协议。
- 网络协议层：其在功能上非常类似于 OSI 模型中的网络层，负责检查网络拓扑结构，以决定传输数据的最佳路由，其最重要的功能是实现 IP 地址和主机的对应（将在下一小节中详细介绍），最常用的协议包括 IP（网际协议）、ICMP（因特网控制消息协议）、ARP（地址解释协议）等。
- 网络接口层：其类似 OSI 模型中的物理层和数据链路层的集合，主要用于实现数据在物理上的传输，即正确的发送和接收 IP 的分组，其涉及的协议和具体的网络相关，例如令牌网、分组交换网的相关协议等。

2. IP 协议规定的 IP 地址

网络（该网络是指宏观的因特网，也即该地址拥有独立 IP）中的任何一台数据设备都必须有一个独一无二的 IP 地址，在 IP 协议中规定了一个 IP 地址由 4 个字节组成，如 159.77.16.17，其对应的二进制为：10011111. 01001101.00010000.00010111。

在 IP 协议中定义了 A、B、C、D 共 4 种主要的地址类。

- A 类地址：第一位固定为 0，第一个字节（前 8 位）为网络标识符，用来标识网络，其余 3 个字节用来标识网络中的主机，因此最多有 127 个 A 类网络，每个 A 类网络可以容纳 1700 万台主机。
- B 类地址：前两位固定为 10，第一个和第二个字节（前 16 位）为网络标识符，用来标识网络，其余 2 个字节用来标识网络中的主机，因此最多有 16000 个 B 类网络，每个 B 类网络可以容纳 65000 台主机。
- C 类地址：前三位固定为 110，前三个字节（前 24 位）为网络标识符，用来标识网络，最后一个字节用来标识网络中的主机，因此最多有 200 万个 C 类网络，每个 C 类网络可以容纳 254 台主机。
- D 类地址：前四位固定为 1110，D 类地址是多目地址，用于标识在网络上运行分布式应用的一群主机，因此，D 类主机并不标识一个在线的主机。

对于一个给定的 IP 地址可以很容易地判别出其地址类别、网络地址和节点地址，例 11.1 是一个对 IP 地址 166.111.111.5 进行判别的实例。

【例 11.1】对 IP 地址进行分类

地址 166.111.111.5 的首字节在 128~19l 之间，因此该地址为 B 类地址，网络地址为 166.111，主机地址为 111.5。

由于 IP 地址由一些数字组成，比较难记住，而记住一个名字则相对来讲容易多了，因此，为了方便使用，必须找到某种机制将网络名称转换成 IP 地址。在 Linux 中，这些名称在文件/etc/hosts 中记录，或者可以要求 DNS（域名服务器）来对名称进行解析。如果由 DNS 来解析地址，那么当地主机必须知道一个或多个 DNS 的 IP 地址，这些 DNS 在文件/etc/resolv.conf 中记录，可以使用 cat 命令来查看本机的该文件（不同的机器可能存在不同），文件的第 3 行输出的是 DNS 服务器的 IP 地址（192.7168.1.1），第 4 行中输出的是本机地址（172.0.0.1）。

```
alloy@ubuntu:~/linuxc/chapter11$ cat /etc/resolv.conf -n
 1  # Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)
 2  #      DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN
 3  nameserver 192.168.1.1
 4  nameserver 127.0.0.1
```

3. Linux 中的 TCP/IP 模型结构

Linux 中的 TCP/IP 模型结构如图 11.4 所示，其是分层模型的良好体现，对各层详细说明如下：

- 在用户态的最上层是各种网络应用程序，包括浏览器、邮件服务器等，使用例如 ftp、ssh 等网络协议。
- 网络应用程序通过 BSD Sockets（BSD 套接字）和 INET Sockets（INET 套接字）这两个套接字接口以及 TCP 或者 UDP 协议进行数据交互，其中 INET 套接字协议还可以直接和 IP 协议进行数据交互。

- TCP 协议和 UDP 协议分别是可靠的有连接的通信协议和不可靠的无连接的通信协议，它们都会和 IP 协议进行数据交互，它们和 IP 协议一起被统称为协议层。
- 在 IP 层或者说协议层下方即为网络接口层，如 PPP、以太网（Ethernet）等，需要注意的是网络设备并不完全等同于物理设备，因为一些网络设备是完全由软件实现的。和其他那些使用 mknod 命令创建的 Linux 系统的标准设备不同，网络设备只有在软件检测到和初始化这些设备时才在系统中出现。当构建系统内核时，即使系统中有相应的以太网设备驱动程序，也只能看到/dev/eth0（网卡）。

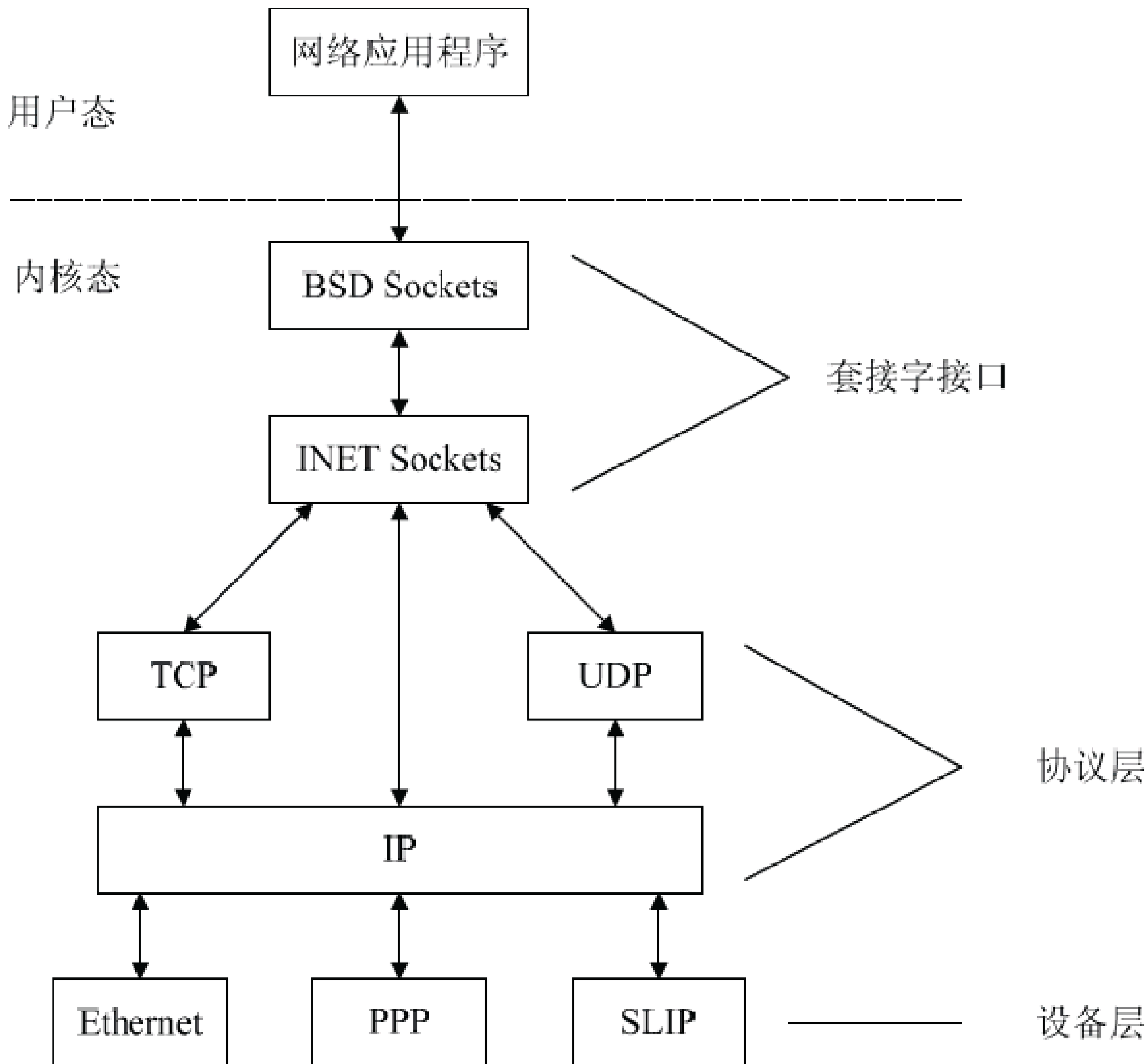


图 11.4　Linux 中的 TCP/IP 模型结构

11.1.3　客户端/服务器结构

TCP/IP 协议允许两个数据设备建立通信并且传输数据，但是其并没有规定这两个数据设备之间数据传输的方法，所以用户需要自行规定一种方法，以达到数据有组织的传输，通常来说会使用客户端/服务器（Client/Server）结构模式来实现，在这种模式下要求这两个数据设备上运行的应用程序，一个作为服务器端存在，而另外一个作为客户端存在，它们的结构如图 11.5 所示，对其功能说明如下：

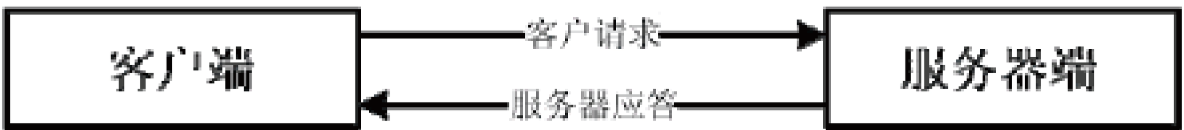


图 11.5　C/S 结构

- 客户端：这是为了得到某种服务所需要运行的应用程序，即申请服务的程序。
- 服务器端：在网络上可提供服务的程序，其接收网络上客户端的请求，完成服务后将结果返回给客户端。

C/S 结构的服务器端通常也可以分为一个主程序和几个从程序，前者负责接收来自客户端的数据，而从程序则负责处理各个客户端的请求，然后交给主程序进行处理，所以服务器端通常可以同

时接收一个或是多个客户的请求，当客户发送某个服务请求时，服务器令其在提供该服务的端口排队，然后从队列中提取请求，为每个请求创建一个子进程，由子进程来处理具体的服务细节，其过程如图 11.6 所示。

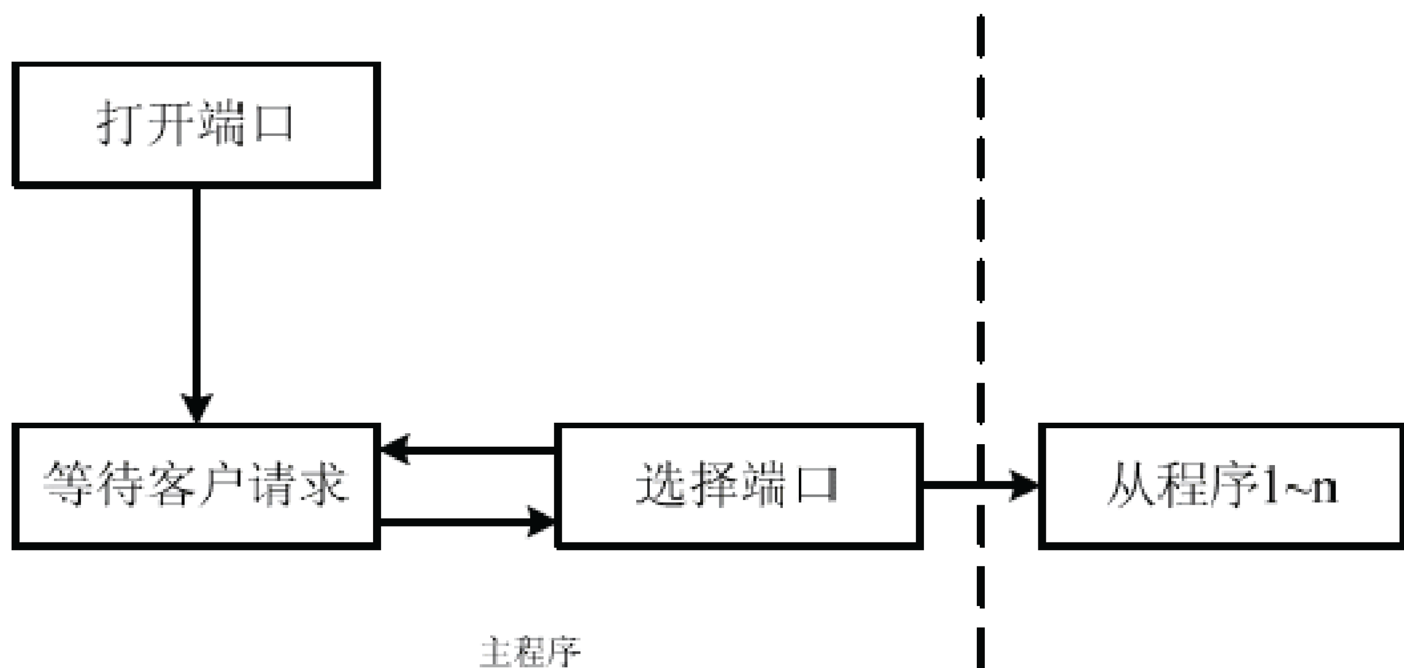


图 11.6　服务器端的主程序和从程序

11.1.4　Linux 的端口和套接字

Linux 的端口是一个逻辑概念，其由 TCP/IP 协议定义，是一个 0~65535 之间的数字，可以分为常用的“固定”端口和通用端口两个部分。所谓的“固定”端口是指一些常用的软件或者 TCP/IP 协议中确定和公布的，通常来说不会被其他程序使用，Linux 中的常见端口和对应的协议如表 11.1 所示。

表 11.1　常用“固定”端口

协议	端口号	协议	端口号
FTP	21	SSH	22
TELNET	23	HTTP	80
TFTP	69	SMTP	25
SNMP	161	DNS	53

所谓套接字（Sockets），即网络进程（服务器端程序或者客户端程序）的进程 ID，和普通的进程 ID 不同，网络进程的 ID 是由运行这个进程的计算机的 IP 地址以及这个进程使用的端口（Port）所组成的，在同一台计算机上一个端口只能分配给一个进程，所以这样就可以确定网络中计算机上的一个进程，套接字的组成如图 11.7 所示。

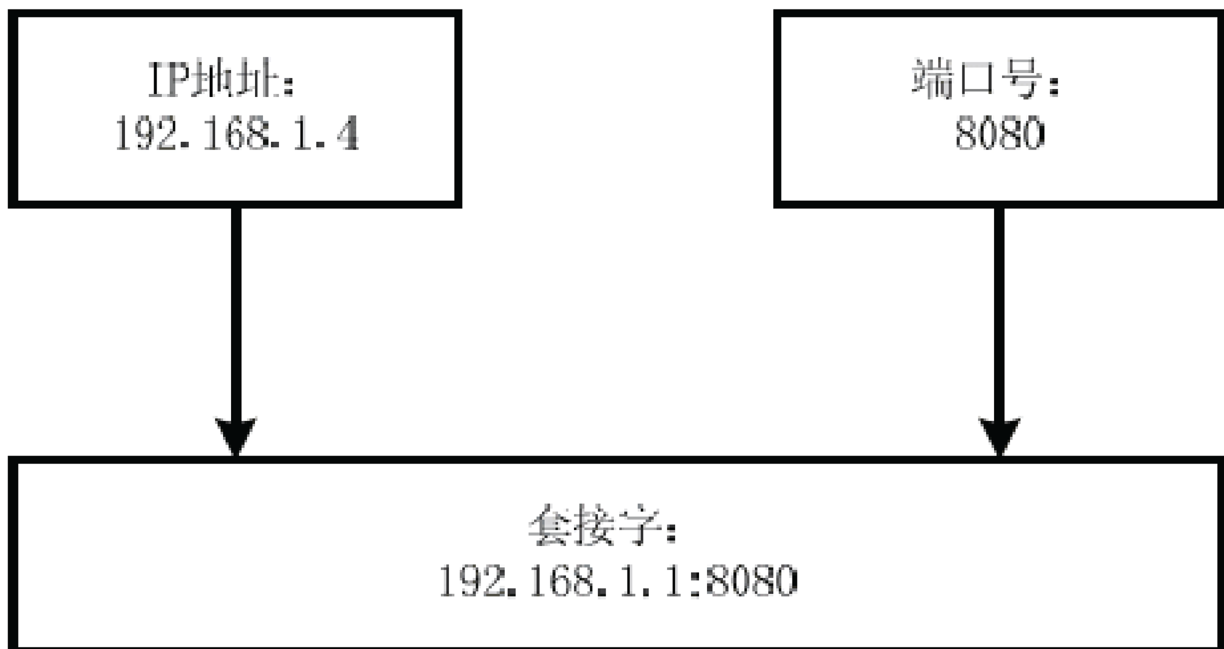


图 11.7　Linux 的套接字组成

可以使用“netstat-all”命令来查看当前系统中网络应用进程的套接字和端口，由于通常来说

该命令的输出会比较多，可以使用“>”命令将输出写入一个文件之后再进行查看，调用如下命令将当前的端口状态写入 netstattest.txt 文件中。

```
alloy@ubuntu:~/linuxc/chapter11$ netstat -all>netstattest.txt
```

由于该文件内容比较多，所以使用 more 命令来分页查看文件内容，首先是激活的 Internet 连接，其中 Proto 表示协议，有 tcp、udp 等；Foreign Address 表示外部地址，而 State 表示状态。

```
alloy@ubuntu:~/linuxc/chapter11$ more netstattest.txt -n
::::::::::::::
netstattest.txt
::::::::::::::
激活 Internet 连接 (服务器和已建立连接的)
Proto Recv-Q Send-Q Local Address          Foreign Address          State
tcp        0      0 localhost:domain       *:*                      LISTEN
tcp        0      0 *:ssh                  *:*                      LISTEN
tcp        0      0 localhost:ipp          *:*                      LISTEN
tcp        1      0 ubuntu.local:60795     mistletoe.canonica:http CLOSE_WAIT
tcp        0     52 ubuntu.local:ssh       alloy-mbp-2.local:51088 ESTABLISHED
tcp6       0      0 [::]:ssh               [::]:*                   LISTEN
udp        0      0 localhost:domain       *:*
udp        0      0 *:bootpc               *:*
udp        0      0 *:bootpc               *:*
udp        0      0 *:45643                *:*
udp        0      0 *:mdns                 *:*
udp6       0      0 [::]:mdns              [::]:*
udp6       0      0 [::]:43321             [::]:*
```

按任意键继续显示文件的内容，可以看到套接字状态，其中 Type 表示套接字的类型，有流和数据报等，I-Node 则表示套接字中对应的端口。

```
活跃的 Unix 域套接字 (服务器和已建立连接的)
Proto RefCnt Flags       Type       State         I-Node   路径
unix  2      [ ACC ]     流         LISTENING     11225    /tmp/.X11-unix/X0
unix  2      [ ACC ]     流         LISTENING     16453    /tmp/keyring-vjclGq/control
unix  2      [ ACC ]     流         LISTENING     12283    @/tmp/.ICE-unix/2374
unix  2      [ ACC ]     流         LISTENING     15702    /tmp/ssh-hslxYFrk2374/agent.2374
unix  2      [ ACC ]     流         LISTENING     12284    /tmp/.ICE-unix/2374
unix  2      [ ACC ]     流         LISTENING     17410    /tmp/keyring-vjclGq/pkcs11
unix  2      [ ACC ]     流         LISTENING     16576    /tmp/keyring-vjclGq/ssh
unix  2      [ ACC ]     流         LISTENING     7482     @/com/ubuntu/upstart
unix  2      [ ACC ]     流         LISTENING     15707    @/tmp/dbus-9evE4bb9VV
unix  2      [ ACC ]     流         LISTENING     8541     @/org/bluez/audio
unix  2      [ ACC ]     流         LISTENING     8534     /var/run/avahi-daemon/socket
unix  2      [ ACC ]     流         LISTENING     8538     /var/run/sdp
unix  2      [ ACC ]     流         LISTENING     11224    @/tmp/.X11-unix/X0
unix  2      [ ACC ]     流         LISTENING     8555     /var/run/cups/cups.sock
unix  17     [ ]         数据报                   7559     /dev/log
unix  2      [ ACC ]     流         LISTENING     9607     /var/run/dbus/system_bus_socket
unix  2      [ ACC ]     流         LISTENING     16073    @/tmp/dbus-T3GyHO3Q
```

Linux 的套接字包括了 BSD 套接字和 INET 套接字两部分，其中 BSD 套接字接口是 Linux 套接字

的基础，从某种意义上说套接字可以看成一种特殊的管道，BSD 套接字通常包括如下几种类型。

- Stream（数据流）：该套接字提供了两个方向的序列数据流，这些数据流保证在传输过程中数据不丢失、不破坏或不重复，数据流套接字由 Internet（INET）地址族的 TCP 协议所支持。
- Datagram（数据报）：该套接字也提供两个方向上的数据传送，但不像数据流套接字，它们不提供消息到达的保证。即使到达也不保证这些数据报按照一定的顺序到达（或丢失、重复）。这种类型的套接字由 Internet 地址族的 UDP 协议所支持。
- Raw（原始套接字）：该套接字允许进程直接访问底层协议。例如，可以为以太网设备打开一个 Raw Socket，以使用原始 IP 数据进行传输。
- Reliable Delivered Message（可靠传递消息）：该套接字非常类似于数据报套接字，但是可保证数据的可靠传输。
- Sequenced Packets（顺序数据报）：这个套接字类似于数据流套接字，但数据包的大小固定。
- Packet（包）：这不是标准的 BSD 套接字类型，它是一个 Linux 特定的扩展，允许进程在设备层直接访问 Packet。

在 Linux 网络编程中最常使用的是支持 TCP 协议的数据流套接字、支持 UDP 协议的数据报套接字和可以直接对底层协议 IP 进行访问的原始格式套接字。

11.1.5　Linux 套接字的结构定义

Linux 在头文件 sys/socket.h 中定义了一种通用的套接字结构类型，以供不同的协议进行调用，对其说明如下：

```
struct sockaddr
  {
    unsigned short int sa_family;          //套接字协议地址类型
    unsigned char sa_data[14];             //14 字节的协议地址，包括 IP 地址和端口
  };
```

对该结构中的分量说明如下。

- sa_family：套接字的协议族地址类型，表 11.2 是常见的协议所对应的 sa_family 值。
- sa_data：具体的协议地址，不同的协议族对应不同的地址结构。

表 11.2　常见协议对应的 sa_family 值

可选值	说明
AF_INET	IPv4 协议
AF_INET6	IPv6 协议
AF_LOCAL	UNIX 协议
AF_LINK	链路地址协议
AF_KEY	密钥套接字

除了 sockaddr 之外，Linux 还在 netinet/in.h 中定义了另外一种结构类型 sockaddr_in，对这种类型的说明如下，其和 sockaddr 等效且可以互相转换，通常来说会在涉及 TCP/IP 的协议编程中使用。

```
struct sockaddr_in
{
        int sa_len:                         //长度单位
        short int sa_family;                //地址族
        unsigned short int sin_port;        //端口号
        struct in_addr sin_addr;            //IP 地址
        unsigned char sin_zero[8];          //填充 0  以保持与 struct sockaddr 同样大小
};
```

对该结构中的各个分量说明如下。

- sa_len：长度单位，不必设置，通常情况下固定长度为 16 字节。
- sa_family：协议族。
- sin_port：端口号
- sin_addr：IP 地址，其本身也是一个结构体，对该结构体的描述说明如下。

```
struct sin_addr
{
    in_addr_t    s_addr;       /*32 位 IPv4 地址，网络字节顺序*/
};
```

- sin_zero：填充 0，目的是为了保持该结构和 sockaddr 结构同样的大小，以方便转换。

在使用结构 sockaddr_in 的时候需要注意以下几点：

- 结构 sockaddr_in 中的 TCP 或 UDP 端口号 sin_port 和 IP 地址 sin_addr 都是以网络字节顺序存储的。
- 32 位的 IP 地址可以利用两种不同的方法引用，例如，假设定义变量 servaddr 为 Internet 套接字的地址结构，那么可以用 servaddr.sin_addr 或 servaddr.sin_addr.s_addr 来引用这个 IP 地址，需要注意的是前一种引用是结构类型（struct sin_addr）的数据，而后一种引用是整数类型的数据；当将 IP 地址作为函数参数使用时，需要明确使用哪种类型的数据，因为编译器对结构类型参数和整数类型参数的处理方式不一样。
- sin_zero 成员未被使用，它是为了和通用套接字地址（struct sockaddr）保持一致而引入的，通常会被填充为 0。
- 套接字地址结构仅供本机 TCP 协议记录套接字信息而用，这个结构变量本身是不在网络上传输的，但是其某些内容，如 IP 地址和端口号是在网络上传输的，这也是为什么这两部分数据需要转换成网络字节顺序的原因。

11.2 Linux 的网络基础操作函数

本节将介绍几个和 Linux 网络编程相关的基础操作函数，包括字节顺序转换函数族、字节操作

函数族、IP 地址转换函数族和域名转换函数族。

11.2.1 字节顺序转换函数族

计算机内部的数据存储通常有两种：大端模式和小端模式，对其说明如下。

- 大端模式：高位字节优先。
- 小端模式：低位字节优先。

通常来说 PC 机中的数据存储采用的是小端模式，某些大型机上的数据存储则采用大端模式，而网络中的数据传输采用的也是大端模式，所以需要对存储的数据进行大小端的转换，图 11.8 给出了大端模式和小端模式的区别。

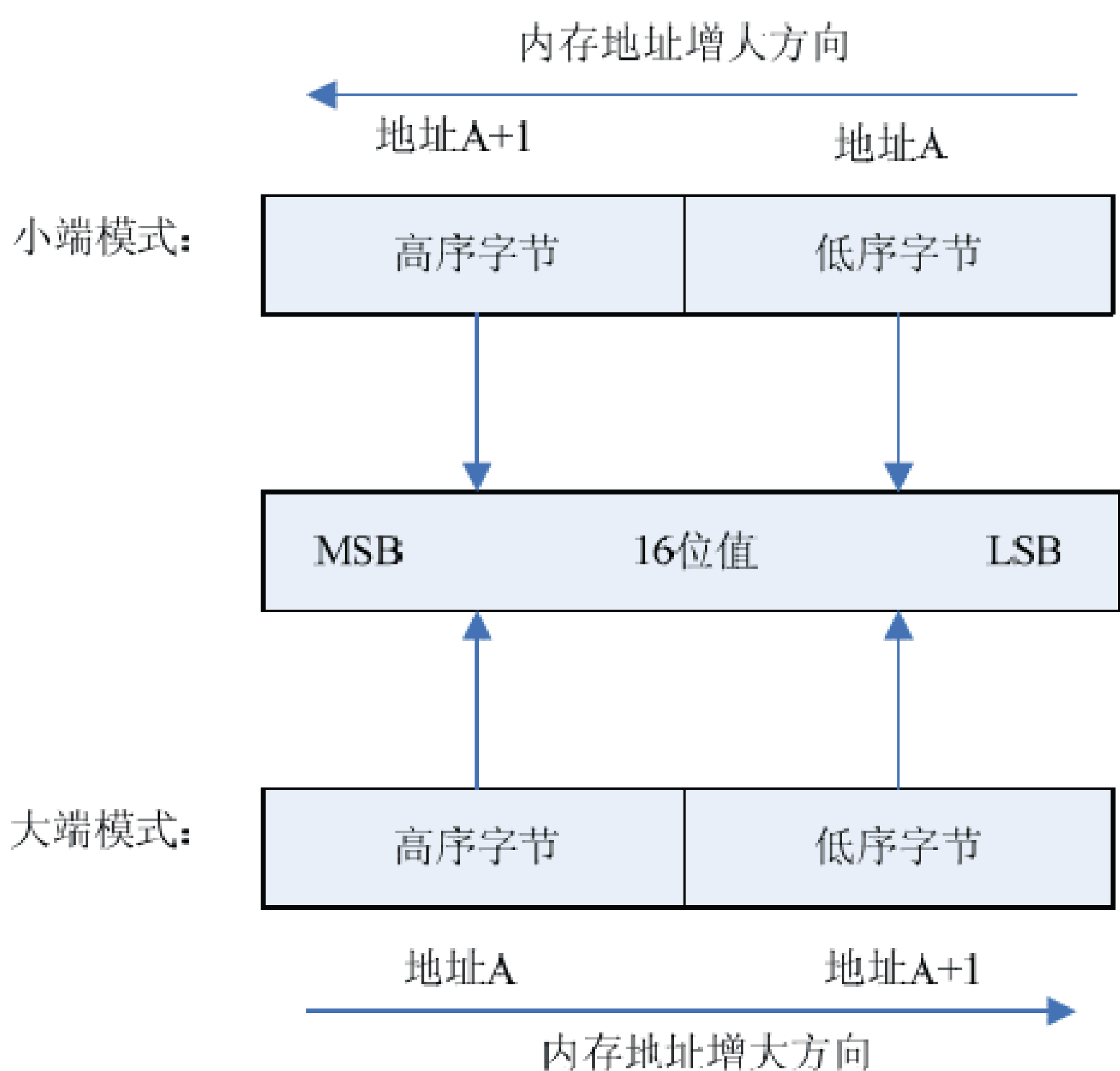


图 11.8 大端模式和小端模式的区别

以 32 位宽度的数据 0x12345678 为例来展示在大端模式和小端模式下的存放方法（假设从内存地址 0x8000 开始存放），如表 11.3 所示。

表 11.3 大端模式和小端模式的数据存放

内存地址	大端模式	小端模式
0x8000	0x12	0x78
0x8001	0x34	0x56
0x8002	0x56	0x34
0x8003	0x78	0x12

Linux 提供了 htonl、htons、ntohl 和 ntohs 这 4 个函数用于处理大端模式和小端模式的数据调换，对其标准调用格式说明如下：

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
```

```
uint16_t ntohs(uint16_t netshort);
```

当函数调用成功返回处理之后得到的值，如果调用失败则返回-1，对函数的功能和参数说明如下。

- htonl 函数：将 32 位的 PC 机数据（小端模式存放）转换为 32 位的网络传输数据（大端模式存放）。
- htons 函数：将 16 位的 PC 机数据（小端模式存放）转换为 16 位的网络传输数据（大端模式存放）。
- ntohl 函数：将 32 位的网络传输数据转换为 32 位的 PC 机数据。
- ntohs 函数：将 16 位的网络传输数据转换为 16 位的 PC 机数据。

以上函数的参数均为对应的需要转换的值，其中 h 代表 host，n 代表 network，s 代表 short，l 代表 long；32 位的 long 数据通常用于存放 IP 地址，而 16 位的 short 数据通常用于存放端口号。

11.2.2 字节操作函数族

由于套接字地址和 C 语言中的字符串不同，为多字节数据而不是以空字符结尾，所以 Linux 提供了两组函数来处理这个多字节数据。

1. 第一组函数

第一组函数是和 BSD 系统兼容的函数，包括了 bzero、bcopy 和 bcmp，对其标准调用格式说明如下。

函数 bzero 将参数 s 指定的内存的前 n 个字节设置为 0，通常用它来将套接字地址清零。

```
#include <strings.h>
void bzero(void *s, size_t n);
```

函数 bcopy 从参数 src 指定的内存区域拷贝指定数目的字节内容到参数 dest 指定的内存区域。

```
#include <strings.h>
void bcopy(const void *src, void *dest, size_t n);
```

函数 bcmp 用于比较参数 s1 指定的内存区域和参数 s2 指定的内存区域的前 n 个字节内容，如果相同则返回 0，否则返回非 0。

```
#include <strings.h>
int bcmp(const void *s1, const void *s2, size_t n);
```

2. 第二组函数

第二组则是标准 C（ANSI C）提供的函数，包括了 memset、memcpy 和 memcmp，对其标准调用格式说明如下：

```
#include <string.h>
```

函数 memset 将参数 s 指定的内存区域的前 n 个字节设置为参数 c 的内容。

```
void *memset(void *s, int c, size_t n);
```

函数 memcpy 和函数 bcopy 的功能相似，两个函数的差别是：函数 bcopy 能处理参数 src 和参数 dest 所指定的区域有重叠的情况，而函数 memcpy 对这种情况没有定义，这时应该使用函数 bcopy。

```
#include <string.h>
void *memcpy(void *dest, const void *src, size_t n);
```

函数 memcmp 用于比较参数 s1 和参数 s2 指定区域的前 n 个字节内容，如果相同则返回 0，否则返回非 0。

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

11.2.3 IP 地址转换函数族

通常来说 IP 地址会被表示为“192.168.1.1”这样的“点分十进制”方式，而在 Linux 的网络编程中会使用 32 位二进制值，所以 Linux 提供了函数族用于将这两个数值进行转换，这些函数包括 inet_aton、inet_addr 和 inet_ntoa 等。

inet_aton 函数用于将点分十进制数的 IP 地址转换成为网络字节序的 32 位二进制数值。输入的点分十进制数 IP 存放在参数 straddr 中，作为返回结果的二进制数值存放在 addrptr 中。

```
#include <arpa/inet.h>
int inet_aton (const char *straddr, struct in_addr *addrptr);
```

与 inet_aton 函数相反，inet_ntoa 函数调用的结果将作为函数的返回值返回给调用它的函数。

```
#include <arpa/inet.h>
char *inet_ntoa (struct in_addr inaddr);
```

int_addr 函数的功能和 inet_aton 函数相同，但是结果传递的方式不同。输入的点分十进制数 IP 仍然存放在参数 straddr 中，但是结果以返回值的形式返回，函数类型为 in_addr_t，不同于 inet_aton 的整型。

```
#include <arpa/inet.h>
in_addr_t inet_addr (const char *straddr);
```

例 11.2 是一个使用 inet_addr 函数将点分十进制数的 IP 地址转换为网络字节序的 32 位二进制数值的实例。

【例 11.2】使用 inet_addr 函数

应用代码将从 argv[1]参数送入的点分十进制字符串中调用 inet_addr 函数，以获得网络字节序的二进制数值，然后将这个数值利用 printf 函数在屏幕上输出。

实例的应用代码如下：

```
1   #include <sys/socket.h>
```

```
 2  #include <netinet/in.h>
 3  #include <arpa/inet.h>
 4  #include <stdio.h>
 5  int main(int argc,char *argv[])
 6  {
 7     unsigned long iptemp;
 8     if(argc != 2)                          //如果参数不正确
 9     {
10        printf("请输入正确的 ip 地址值.\n");
11        return 1;
12     }
13     iptemp = inet_addr(argv[1]);           //调用 inet_addr 函数获得网络地址
14     printf("返回的 ip 数值是%lu.\n",iptemp);
15     return 0;
16  }
```

将文件保存为 exam1101inetaddr.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam1101inetaddr。

```
alloy@ubuntu:~/linuxc/chapter11$ gcc exam1101inetaddr.c -o exam1101inetaddr
```

执行该可执行文件，将 IP 地址 192.168.1.1 作为待转换的参数，可以看到返回的网络 IP 地址为 16885952。

```
alloy@ubuntu:~/linuxc/chapter11$ ./exam1101inetaddr 192.168.1.1
返回的 ip 数值是 16885952.
```

在实际应用中应该避免使用 inet_addr 函数，而应该使用 inet_aton 函数代替。因为对于 inet_addr 函数来说，即使输入的参数是有效的 IP 地址：255.255.255.255，它的返回值仍然是 INADDR_NONE。INADDR_NONE 是 Linux 下定义的一个常量，表示一个不存在的 IP 地址，当返回这个常数时，就说明转换出了问题。一般将这个常量定义成 255.255.255.255（对应 Internet 的有限广播地址），利用二进制表示再转换成有符号数，则为-1，另外此时并没有在 Linux 系统中建立一个实际的 error 值，所以不应该对这个值进行处理。

例 11.3 是使用 inet_aton 函数实现例 11.2 功能的实例。

【例 11.3】使用 inet_aton 函数

应用代码依然将 argv[1]传递的点分十进制 IP 地址交给 inet_aton 函数进行处理，并且返回函数处理后得到的网络地址，其和 inet_addr 函数不同，其返回值是一个 in_addr 结构的结构体，所以预先定义了一个 in_addr 结构的结构体变量 testaddr，传递给 inet_aton 函数用以获得对应的网络地址。

实例的应用代码如下：

```
1  #include <sys/socket.h>
2  #include <netinet/in.h>
3  #include <arpa/inet.h>
4  #include <stdio.h>
5  int main(int argc,char *argv[])
```

```
6   {
7       int temp;
8       struct in_addr *testaddr;    //定义一个结构体
9       if(argc != 2)
10      {
11          printf("请输入正确的 ip 地址.\n");
12          return 1;
13      }
14      temp = inet_aton(argv[1],testaddr);
15      if(temp == 0)
16      {
17          printf("调用 inet_aton 失败.\n");
18          return 1;
19      }
20      else
21      {
22          printf("转换后的 IP 地址是%lu.\n",testaddr->s_addr);
23      }
24      return 0;
25  }
```

将文件保存为 exam1102inetaton.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam1102inetaton。

```
alloy@ubuntu:~/linuxc/chapter11$ gcc exam1102inetaton.c -o exam1102inetaton
```

依然对 IP 地址 192.168.1.1 执行该可执行文件，可以看到转换之后的网络地址数值还是 16885952。

```
alloy@ubuntu:~/linuxc/chapter11$ ./exam1102inetaton 192.168.1.1
转换后的 ip 地址是 16885952.
```

和例 11.2、例 11.3 相反，例子 11.4 是一个使用 inet_ntoa 函数的实例，其展示了如何将网络地址转换为对应的点分十进制 IP 地址。

【例 11.4】使用 inet_ntoa 函数

应用代码首先使用 inet_addr 函数将通过 argv[1]和 argv[2]参数传递的两个点分十进制 IP 地址转换为对应的网络地址，由于 inet_ntoa 函数的参数必须是 in_addr 类型的结构体变量，所以使用 memcpy 函数将网络地址直接拷贝到两个 in_addr 类型的结构体变量 netaddr1 和 netaddr2 之后，再传递给 inet_ntoa 函数进行处理，最后在屏幕上输出。

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <sys/socket.h>
3   #include <netinet/in.h>
4   #include <arpa/inet.h>
5   #include <string.h>
6   int main(int argc, char *argv[])
```

```
7   {
8     struct in_addr addr1,addr2;
9     unsigned long netaddr1,netaddr2;
10    if(argc != 3)                                    //如果参数不正确
11    {
12      printf("请输入正确的参数.\n");
13      return 1;                                      //退出
14    }
15    netaddr1 = inet_addr(argv[1]);
16    netaddr2 = inet_addr(argv[2]);
17    memcpy(&addr1, &netaddr1, 4);
18    memcpy(&addr2, &netaddr2, 4);                   //拷贝地址
19    printf("addr1 = %s : addr2 = %s\n", inet_ntoa(addr1), inet_ntoa(addr2));
      //再次输出两个 IP 地址
20    //分别输出 IP 地址
21    printf("%s\n", inet_ntoa(addr1));
22    printf("%s\n", inet_ntoa(addr2));
23    return 0;
24  }
```

将文件保存为 exam1103inetntoa.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam1103inetntoa。

```
alloy@ubuntu:~/linuxc/chapter11$ gcc exam1103inetntoa.c -o exam1103inetntoa
```

将 IP 地址 192.168.1.1 和 211.21.32.23 传递给可执行文件进行处理，可以看到对应的输出，其中第一行中连续输出了两次 192.168.1.1，这是因为 inet_ntoa 函数的返回值直到下次调用前一直有效，所以如果在线程中使用 inet_ntoa 的时候，一定要确保每次只有一个线程调用本函数，否则一个线程返回的结构可能被其他线程返回的结果所覆盖。

```
alloy@ubuntu:~/linuxc/chapter11$ ./exam1103inetntoa 192.168.1.1 211.21.32.23
addr1 = 192.168.1.1 : addr2 = 192.168.1.1
192.168.1.1
211.21.32.23
```

11.2.4 域名转换函数族

在实际的网络应用中，常常会使用类似“www.sina.com.cn”这样的域名替代 IP 地址来标识一个服务器，所以需要一个函数将这个域名转换为实际的 IP 地址，还需要一个函数能将实际的 IP 转换为域名。

Linux 在 netdb.h 头文件中定义了一个结构体，用于描述一个主机的相关参数，其形式如下：

```
struct hostent
{
  char *h_name;                    //主机的正式名称
  char *h_aliases;                 //主机的别名
  int h_addrtype;                  //主机的地址类型，IPv4 为 AF_INET
  int h_length;                    //主机的地址长度，对于 IPv4 是 4 字节，即 32 位
```

```
    char **h_addr_list;                    //主机的 IP 地址列表
};
#define h_addr h_addr_list[0]              //主机的第一个 IP 地址
```

Linux 提供了 gethostbyname 和 gethostbyaddr 函数用于处理域名和地址的转换，对其标准调用格式说明如下：

```
#include <netdb.h>
extern int h_errno;
struct hostent *gethostbyname(const char *name);
#include <sys/socket.h>
struct hostent *gethostbyaddr(const void *addr,socklen_t len, int type);
```

gethostbyname 函数用于实现域名或主机名到 IP 地址的转换，参数 hostname 指向存放域名或主机名的字符串。

gethostbyaddr 函数用于实现 IP 地址到域名或主机名的转换，参数 addr 是一个指向含有地址结构（in_addr 或 in_addr）的指针；参数 len 是此结构的大小，对于 IPv4 而言其值为 4，对于 IPv6 而言其值为 16，参数 family 为协议族。

若调用这两个函数成功，则返回一个指向 hostent 结构的指针，若调用失败则返回空指针 NULL，同时设置全局变量 h_errno 为相应的值，h_errno 可能的取值如表 11.4 所示。

表 11.4 h_errno 可能的取值

h_errno	说明
HOST_NOT_FOUND	找不到对应的主机
TRY_AGAIN	出错重试
NO_RECOVERY	出现了不可修复的错误
NO_DATA	该名字有效，但是没有找到该记录

例 11.5 是一个使用 gethostbyname 函数来获取指定域名对应 IP 地址的实例。

【例 11.5】使用 gethostbyname 函数

应用代码首先定义了一个地址结构体 hpaddr 和一个 hostent 类型的指针 hptr，使用 gehostbyname 函数获取 arvg[1]指定参数的 IP 地址，然后使用 memcpy 将其复制到结构体 hpaddr 中，这是因为 inet_ntoa 需要一个地址类型的结构体作为参数，最后使用 printf 函数输出 inet_ntoa 函数的转换值。

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/types.h>
4   #include <netdb.h>
5   #include <sys/socket.h>
6   #include <string.h>
7   #include <arpa/inet.h>
8   int main(int argc,char *argv[])
9   {
```

```
10    struct hostent *hptr;
11    struct in_addr hpaddr;                               //定义一个地址结构体
12    if((hptr = gethostbyname(argv[1])) == NULL)
13    {
14      printf("请输入域名.\n");
15      return 1;
16    }
17    else
18    {
19      memcpy(&hpaddr,&hptr->h_addr,4);                  //复制 IP 地址
20      printf("IP 地址为%s.\n",inet_ntoa(hpaddr));
21    }
22    return 0;
23  }
```

将文件保存为 exam1104gethost.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam1104gethost。

```
alloy@ubuntu:~/linuxc/chapter11$ gcc exam1104gethost.c -o exam1104gethost
```

对 www.263.net 域名执行该可执行文件，可以看到如下的 IP 地址输出：

```
alloy@ubuntu:~/linuxc/chapter11$ ./exam1104gethost www.263.net
IP 地址为 204.161.225.1.
```

例 11.6 是一个功能更加复杂的 gethostbyname 函数的使用方法，其不仅仅输出了域名对应的 IP 地址，还输出了域名对应的正式地址和别名。

【例 11.6】使用 gethostbyname 函数

应用代码的基础操作和例 11.5 完全相同，只是将调用 gethostbyname 获得域名对应的数据都放入了 hptr 所指向的结构体之后，分别将该结构体的 h_name 和 h_aliases 分量也输出，由于 h_aliases 的分量可能不止一个，所以使用了 for 循环。

实例的应用代码如下：

```
 1  #include <netdb.h>
 2  #include <sys/socket.h>
 3  #include <stdio.h>
 4  #include <stdlib.h>
 5  #include <sys/types.h>
 6  #include <string.h>
 7  #include <arpa/inet.h>
 8  #include <netinet/in.h>
 9  int main(int argc,char *argv[])
10  {
11    char *ptr,**pptr;
12    struct hostent *hptr;
13    struct in_addr hpaddr;
14    //使用 argv[1]作为参数
```

```
15     ptr = argv[1];
16     //调用 gethostbyname 函数，将结果存放到 hptr 中
17     if((hptr = gethostbyname(ptr))== NULL)              //如果调用函数失败
18     {
19       printf("解析域名%s 失败.\n",ptr);
20       return 0;
21     }
22     printf("目标的官方域名是%s\n",hptr->h_name);
23     //由于目标可能有多个别名，所以全部打印
24     for(pptr = hptr->h_aliases;*pptr != NULL;pptr++)
25     {
26       printf("目标的别名是%s\n",*pptr);
27     }
28     //根据地址类型输出地址
29     switch(hptr->h_addrtype)
30     {
31       case AF_INET:
32       case AF_INET6:    //针对 IPv4 和 IPv6 均进行如下操作，因为之前没有 break
33       {
34         pptr = hptr -> h_addr_list;
35         for(;*pptr != NULL;pptr++)
36         {
37           memcpy(&hpaddr,pptr,4);
38           printf("目标地址是:%s\n",inet_ntoa(hpaddr));
39         }
40       }
41       break;
42       default:
43         printf("未知的地址类型\n");
44     }
45     return 0;
46   }
```

将文件保存为 exam1105gethostbyname.c，在终端中使用 gcc 进行编译链接，生成可执行文件。

```
alloy@ubuntu:~/linuxc/chapter11$ gcc exam1105gethostbyname.c -o exam1105gethostbyname
```

对 www.sina.com.cn 域名执行该可执行文件，可以看到如下的输出，这个域名有两个别名：

```
alloy@ubuntu:~/linuxc/chapter11$ ./exam1105gethostbyname www.sina.com.cn
目标的官方域名是 polaris.sina.com.cn
目标的别名是 www.sina.com.cn
目标的别名是 jupiter.sina.com.cn
目标地址是:52.242.174.0
```

11.3 Linux 的网络套接字操作函数

套接字编程是 Linux 的网络编程基础，本节将介绍一些与其相关的函数以及其应用方法。

1．创建套接字描述符函数

Linux 使用 socket 函数来创建一个套接字描述符，对该函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

如果函数调用成功，则返回套接字的描述符，这是一个正整数，如果函数调用失败则返回-1，对函数中的各个参数描述如下。

- domain：套接字的协议族，其支持的类型说明如表 11.5 所示，socket 函数可以支持多种网络协议，在使用的时候必须指定当前使用的协议。

表 11.5　socket 函数支持的协议族

协议族名称	描述
AF_UNIX，AF_LOCAL	本地交互协议
AF_INET	IPv4 协议
AF_INET6	IPv6 协议
AF_IPX	IPX-Novell 协议
AF_NETLINK	内核接口设备协议
AF_X25	ITU-T X.25/ISO-8208 协议
AF_AX25	业余无线电 AX.25 协议
AF_ATMPVC	原始 ATM 接入协议
AF_APPLETALK	苹果的 Appletalk 协议
AF_PACKET	底层数据包接口

- type：用于指定当前的套接字类型，socket 函数支持的套接字类型包括 SOCK_STREAM（数据流）、SOCK_DGRAM（数据报）、SOCK_SEQPACKET（顺序数据报）、SOCK_RAW（原始套接字）、SOCK_RDM（可靠传递消息）、SOCK_PACKET（数据包）。
- protocol：除了在使用原始套接字以外，通常情况下设置为 0，以表示使用默认的协议。

在 Linux 系统中创建一个套接字时会在内核中创建一个套接字数据结构，然后返回一个套接字描述符标识这个套接字数据结构。这个套接字数据结构包含连接的各种信息，如对方地址、TCP 状态以及发送接收缓冲区等。TCP 协议根据这个套接字数据结构的内容来控制这条连接。

例 11.7 是使用 socket 函数来创建一个 TCP 套接字的实例。

【例 11.7】使用 socket 函数创建套接字

应用代码调用 socket 函数来建立了一个套接字，设定套接字使用的协议是 AF_INET，即 IPv4，套接字类型为 STREAM，如果创建成功，则返回对应的套接字描述符。

实例的应用代码如下：

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3  #include <stdio.h>
4  int main(int argc,char *argv[])
5  {
6      int sockfd;                                          //定义套接口描述符
7      if((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)       //建立一个 socket
8      {
9          printf ("创建套接字失败.\n");
10         return 1;
11     }
12     else        //socket 创建成功
13     {
14         printf("套接字的 ID 是:%d\n",sockfd);
15     }
16     return 0;
17 }
```

将文件保存为 exam1106socket.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam1106socket。

```
alloy@ubuntu:~/linuxc/chapter11$ gcc exam1106socket.c -o exam1106socket
```

执行该可执行文件，可以看到对应的套接字描述符输出。

```
alloy@ubuntu:~/linuxc/chapter11$ ./exam1106socket
套接字的 ID 是:3
```

2. 绑定套接字函数

在创立了套接字之后需要将本地地址和套接字绑定在一起，此时可以调用 bind 函数，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr,socklen_t addrlen);
```

其中参数 sockfd 是使用 socket 函数创建的套接字对应的套接字描述符，addr 是本地地址，addrlen 是套接字对应的地址结构长度；如果 bind 函数执行成功，则返回 0，否则返回-1。

在第 11.1.3 小节中介绍了 Linux 的客户端和服务器模式，在网络通信中服务器和客户端都可以使用 bind 函数来设置套接字地址，通常来说有以下 5 种模式：

- 服务器指定套接字地址的公认端口号，不指定 IP 地址，服务器调用函数 bind 时，如果设置套接字的 IP 地址为特殊的 INADDR_ANY，表示它愿意接收来自任何网络设备接口的客户端连接，这是服务器最经常使用的绑定方式。
- 服务器指定套接字地址的公认端口号和 IP 地址，服务器调用函数 bind 时，如果设置套

接字的 IP 地址为某个本地 IP 地址，则表示服务器只接收来自对应于这个 IP 地址的特定网络设备接口的客户端连接。如果这台机器只有一个网络设备接口，则这和第一种情况是没有区别的，但当这台机器有多个网络设备接口时，我们可以用这种方式来限制服务器的接收范围。

- 客户端指定套接字地址的连接端口号，在一般情况下，客户端不用指定自己的套接字地址的端口号，当客户端调用函数 connect 进行 TCP 连接时，系统会自动为它选择一个未用的端口号，并且用本地的 IP 地址来填充套接字地址中的相应项，但在有的情况下，客户端需要使用特定端口号。
- 指定客户端的 IP 地址和连接端口号，表示客户端使用指定的网络设备接口和端口号进行通信。
- 指定客户端的 IP 地址，表示客户端使用指定的网络设备接口进行通信，系统自动为客户端选择一个未用的端口号。一般情况下，只有在主机有多个网络设备接口时使用。

对以上组合的总结如表 11.6 所示。

表 11.6　使用 bind 函数对应的组合方式

C/S	IP	port	说明
服务器	INADDR_ANY	非 0 值	指定服务器的公认端口号
服务器	本地 IP 地址	非 0 值	指定服务器的 IP 地址和公认端口号
客户端	INADDR_ANY	非 0 值	指定客户端的连接端口号
客户端	本地 IP 地址	非 0 值	指定客户端的 IP 地址和连接端口号
客户端	本地 IP 地址	0	指定客户端的 IP 地址

在编写客户端程序时，通常不要使用固定的客户端端口号，除非是在必须使用特定端口的情况下，因为固定客户机端口号会带来一些不便，例如如下两种情况。

- 服务器执行主动关闭操作：服务器最后进入 TIME_WAIT 状态。当客户机再次与这个服务器进行连接时，仍使用相同的客户机端口号，于是这个连接与前次连接的套接字对完全相同，但是因为前次连接处于 TIME_WAIT 状态，并未消失，所以这次连接请求被拒绝，函数 connect 以错误返回。
- 客户端执行手动关闭操作：客户端最后进入 TIME_WAIT 状态，当立刻再次执行这个客户机程序时，客户机将继续与这个固定客户机端口号绑定，但因为前次连接处于 TIME_WAIT 状态，并未消失，系统会发现这个端口号仍被占用，所以这次绑定操作失败，函数 bind 以错误返回。

例 11.8 是一个使用 bind 函数绑定套接字的实例。

【例 11.8】使用 bind 函数绑定套接字

应用代码定义了一个 IPv4 的套接字地址数据结构变量 addr，首先使用 socket 函数创建一个套

接字，然后使用 bzero 函数将结构变量 addr 的值清空，分别设置结构体的各个分量，最后调用 bind 函数将这个变量绑定到刚刚创建的套接字上。

实例的应用代码如下：

```
 1  #include <sys/types.h>
 2  #include <sys/socket.h>
 3  #include <netinet/in.h>
 4  #include <arpa/inet.h>
 5  #include <unistd.h>
 6  #include <stdio.h>
 7  #include <string.h>
 8  #define   PORT 5555                                          //定义端口号
 9  int main(int argc,char *argv[])
10  {
11      int sockfd;                                              //定义套接口描述符
12      struct sockaddr_in addr;                                 //定义 IPv4 套接口地址数据结构 addr
13      int addr_len = sizeof(struct sockaddr_in);
14      if((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)          //建立一个 socket
15      {
16          printf("创建套接字失败!\n");
17          return 1;
18      }
19      bzero(&addr,sizeof(struct sockaddr_in));                 //清空表示地址的结构体变量
20      addr.sin_family = AF_INET;                               //设置 addr 的成员信息
21      addr.sin_port = htons(PORT);
22      addr.sin_addr.s_addr = htonl(INADDR_ANY);                //IP 地址设为本机 IP
23      if(bind(sockfd, (struct sockaddr *)(&addr), sizeof(struct sockaddr))<0)
24      {
25          printf("绑定端口失败!");
26          return 1;
27      }
28      return 0;
29  }
```

将文件保存为 exam1107bind.c，在终端中使用 gcc 进行编译链接，生成可执行文件，需要注意的是这个可执行文件没有输出。

```
alloy@ubuntu:~/linuxc/chapter11$ gcc exam1107bind.c -o exam1107bind
```

3. 建立连接函数

当使用 socket 函数建立一个套接字并且绑定了地址之后，即可使用 connect 函数来和服务器建立一个连接，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr,socklen_t addrlen);
```

其中参数 sockfd 是套接字创建函数 socket 返回的套接字描述符，参数 addr 指定远程服务器的

套接字地址，包括服务器的 IP 地址和端口号；参数 addrlen 指定这个套接字地址的长度；当调用成功后函数返回 0，否则返回-1。

在调用 connect 函数建立连接之前，客户端应用代码需要指定服务器端进程的套接字地址，而客户端通常不会指定自己的套接字地址，Linux 会自动从 1024~5000 的端口范围之中为客户端分配一个未被使用的端口号，然后将该端口号和本机的 IP 地址结合在一起放入套接字地址中。

当客户端调用函数 connect 来主动建立连接时，这个函数将启动 TCP 协议的 3 次握手过程（参考第 11.4 节），在连接建立之后或发生错误时，函数返回。连接过程中可能有如下几种错误情况：

- 如果客户机 TCP 协议没有接收到对它的 SYN 数据段确认，则函数以错误返回，错误类型为 ETIMEOUT。通常情况下，TCP 协议在发送 SYN 数据段失败之后，会多次发送 SYN 数据段，在所有的发送都宣告失败之后，函数以错误返回。
- 如果远程 TCP 协议返回一个 RST 数据段，则函数立即以错误返回，错误类型为 ECONNREFUSED。当远程机器在 SYN 数据段指定的目的端口号处没有服务器进程在等待连接时，远程机器的 TCP 协议将发送一个 RST 数据段，向客户机报告这个错误。客户机的 TCP 协议在接收到 RST 数据段之后，不再继续发送 SYN 数据段，函数立即以错误返回。
- 如果客户机的 SYN 数据段导致某个路由器产生“目的地不可到达”类型的 ICMP 消息，则函数以错误返回，错误类型为 EHOSTUNREACH 或 ENETUNREACH。通常情况下，TCP 协议在接收到这个 ICMP 消息之后，记录这个消息，然后继续几次发送 SYN 数据段，在所有的发送都宣告失败之后，TCP 协议检查这个 ICMP 消息，函数以错误返回。

如果调用函数 connect 失败，应该用函数 close 关闭这个套接字描述符，不能再次用这个套接字描述符来调用函数 connect。

例 11.9 是一个使用 connect 函数来建立连接的实例。

【例 11.9】使用 connect 函数建立连接

应用代码使用 PORT 和 REMOTE_IP 来分别定义一个端口号和一个 IP 地址，然后分别调用 socket 和 bind 函数来创建套接字和绑定套接字，最后使用 connect 函数来连接 argv[1]参数中所指定的 IP 地址。

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <netinet/in.h>
3   #include <arpa/inet.h>
4   #include <unistd.h>
5   #include <fcntl.h>
6   #include <sys/stat.h>
7   #include <sys/types.h>
8   #include <sys/socket.h>
9   #include <string.h>
10  #define PORT 80                                   //定义一个端口号
11  //#define REMOTE_IP "59.175.132.70"               //定义一个 IP 地址
```

```
12  int main(int argc,char *argv[])
13  {
14      int sockfd;
15      struct sockaddr_in addr;                          //定义 IPv4 套接字地址数据结构 addr
16      if(argc != 2)
17      {
18        printf("请输入正确的 ip 地址字符串.\n");
19        return 2;
20      }
21      if( (sockfd = socket(AF_INET,SOCK_STREAM,0))<0 )      //建立一个 socket
22      {
23          printf("创建套接字失败!\n");
24          return 1;
25      }
26      bzero(&addr,sizeof(struct sockaddr_in));              //清空表示地址的结构体变量
27      addr.sin_family = AF_INET;                            //设置 addr 的成员信息
28      addr.sin_port = htons(PORT);
29      addr.sin_addr.s_addr = inet_addr(argv[1]);            //从 argv[1]中获得目标的 IP 地址
30      if(connect(sockfd, (struct sockaddr *)(&addr), sizeof(struct sockaddr))<0)
31      {
32          printf("连接失败!\n");
33          return;
34      }
35      else
36      {
37          printf("连接成功!\n");
38      }
39  return 0;
40  }
```

将文件保存为 exam1108connect.c，在终端中使用 gcc 进行编译链接，生成可执行文件，

```
alloy@ubuntu:~/linuxc/chapter11$ gcc exam1108connect c -o exam1108connect
```

对 192.168.1.1 执行该可执行文件，可以看到连接成功：

```
alloy@ubuntu:~/linuxc/chapter11$ ./exam1108connect 192.168.1.1
连接成功!
```

此时可以使用“netstat -nap|grep”命令来查看对 192.168.1.1 的 80 端口的连接状态：

```
alloy@ubuntu:~/linuxc/chapter11$ sudo netstat -nap|grep '192.168.1.1:80'
tcp        0      0 192.168.1.5:55828       192.168.1.1:80          TIME_WAIT    -
```

4. 倾听套接字切换函数

对于服务器端的应用程序而言，在创立了套接字之后通常需要等待客户端的连接，此时可以使用 listen 函数将该套接字转换为倾听套接字，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int listen(int sockfd, int backlog);
```

其参数 sockfd 为待转换的套接字描述符，参数 backlog 为设置请求队列的最大长度，当调用该函数成功时返回 0，若调用失败则返回-1。

服务器需要调用函数 listen 将套接字转换成倾听套接字，以便接收客户机请求。函数 listen 的功能有如下两个：

- 函数 socket 创建的套接字是主动套接字，可以用它来进行主动连接（调用函数 connect），但是不能接收连接请求，而服务器的套接字必须能够接收客户机的请求。函数 listen 将一个尚未连接的主动套接字转换成为一个被动套接字，并告诉 TCP 协议，这个套接字可以接收连接请求。
- TCP 协议将到达的连接请求排队，函数 listen 的第 2 个参数指定这个队列的最大长度。

若要创建一个倾听套接字，必须首先调用函数 socket 创建一个主动套接字，然后调用函数 bind 将它与服务器套接字地址绑定在一起，最后调用函数 listen 进行转换。这 3 个操作是所有 TCP 服务器必须执行的操作。

下面讨论参数 backlog 的作用，这对于理解套接字建立连接的过程非常重要，TCP 协议为每个倾听套接字维护两个队列。

- 未完成连接队列：每个尚未完成 3 次握手操作的 TCP 连接在这个队列中占有一项。TCP 协议在接收到一个客户机 SYN 数据段之后，在这个队列中创建一个新条目，然后发送对客户机 SYN 数据段的确认以及自己的 SYN 数据段（ACK+SYN 数据段），等待客户机对自己的 SYN 数据段确认：此时，套接字处于 SYN_RCVD 状态，这个条目将保存在队列中，直到客户机返回对 SYN 数据段的确认，或者连接超时。
- 完成连接队列：每个已经完成 3 次握手操作，但尚未被应用程序接收（调用函数 accept）的 TCP 连接在这个队列中占有一项。当一个在未完成连接队列中的连接接收到对 SYN 数据段的确认之后，完成 3 次握手操作，TCP 协议将它从未完成的连接队列中移到完成连接队列中。这个条目将保存在队列中，直到应用程序调用函数 accept 来接收它。

参数 backlog 指定倾听套接字的完成连接队列的最大长度，表示这个套接字能够接收的最大数目的未接收（unaccepted）连接。如果当一个客户机的 SYN 数据段到达时，倾听套接字的完成连接队列已经满了，那 TCP 协议将忽略这个 SYN 数据段。对于不能接收的 SYN 数据段，TCP 协议不发送 RST 数据段，原因有两个：

- 假设 TCP 协议在未完成队列满时返回 RST 数据段，那么客户机的函数 connect 将马上以错误返回，不再继续发送连接请求。根据这个 RST 数据段，客户机无法知道，究竟是这个端口上没有服务器进程在等待连接，还是在这个端口上等待的服务器的未完成连接队列暂时没有空间。
- 完成队列满的情况是暂时的：经过一段时间之后，应用程序可能调用函数 accept 从这个完成队列中接收已经建立的连接，于是完成队列中出现新的空间。客户机 TCP 协议在超时之后，继续发送几次 SYN 数据段。如果在这几次发送过程中，完成连接队列中出

现新的空间，那么 TCP 协议将接收这个连接请求，继续正常的 3 次握手操作。如果在这几次发送过程中，完成连接队列中都没有空间，客户机将放弃发送。


注 意

TCP 协议下的数据握手发送方式将在第 11.4 节中详细介绍。

5. 接收连接函数

当服务器端倾听到一个连接之后，可以使用函数 accept 从倾听套接字的完成连接队列中接收一个连接，如果这个完成连接队列为空，则会使得这个进程进入睡眠状态，对 accept 函数的标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

参数 sockfd 指定套接字描述符；参数 addr 为指向一个 Internet 套接字地址结构的指针；参数 addrlen 为指向一个整型变量的指针。当函数 accept 成功执行时，返回 3 个结果：

- 函数返回值为一个新的套接字描述符，标识这个接收的连接。
- 参数 addr 指向的结构变量中存储客户机地址。
- 参数 addrlen 指向的整型变量中存储客户机地址的长度。

如果对客户机的地址和长度都不感兴趣，可以将参数 addr 和 addrlen 设置为 NULL，若函数 accept 执行失败时，返回-1。

函数 accept 从倾听套接字的完成连接队列中接收一个已经建立起来的 TCP 连接，因为倾听套接字是专为接收客户机连接请求，完成 3 次握手操作而用的，所以 TCP 协议不能使用倾听套接字描述符来标识这个连接，于是 TCP 协议创建一个新的套接字来标识这个要接收的连接，并将它的描述符返回给应用程序。现在有两个套接字：一个是调用函数 accept 时使用的倾听套接字，另一个是函数 accept 返回的连接套接字（connected socket）。这两个套接字的作用是完全不同的：一个服务器进程通常只需创建一个倾听套接字，在服务器进程的整个活动期间，用它来接收所有客户机的连接请求，在服务器进程终止前关闭这个倾听套接字；而对于每个接收的（accepted）连接，TCP 协议都创建一个新的连接套接字来标识这个连接，服务器使用这个连接套接字与客户机进行通信操作，当服务器处理完这个客户机请求时，关闭这个连接套接字。

当函数 accept 阻塞等待已经建立的连接时，如果进程捕获到信号，那么函数将以错误返回，错误类型为 EINTR。对于这种错误，一般情况下重新调用函数 accept 来接收连接。

6. 关闭连接函数

当操作完成之后，可以使用 close 函数来关闭当前建立的连接，对其标准调用格式说明如下，需要注意的是其和前面中介绍的文件操作函数 close 的名称相同，但是属于 unistd.h 头文件，其参数也不是文件描述符 fd，而是套接字描述符 sockfd，如果函数调用成功，则返回 0，否则返回-1。

```
#include <unistd.h>
int close(int fd);
```

套接字描述符的 close 操作和文件描述符的 close 操作一样：函数 close 将套接字描述符的引用计数减 1，如果描述符的引用计数大于 0，则表示还有进程引用这个描述符，函数 close 正常返回；如果描述符的引用计数变为 0，则表示再没有进程引用这个描述符，于是启动清除套接字描述符的操作，函数 close 立即正常返回。清除套接字描述符的操作是：将这个套接字描述符标记为关闭状态，然后立即返回进程。调用了函数 close 之后，进程将不再能够访问这个套接字，但是这不表示 TCP 协议删除了这个套接字。TCP 协议将继续使用这个套接字，将尚未发送的数据传递到对方，然后发送 FIN 数据段，执行关闭操作，一直等到这个 TCP 连接完全关闭之后，TCP 协议才删除这个套接字。

7. 套接字读写函数

函数 read/write 用于从套接字读/写数据。其定义如下：

```
int read(int fd, char *buf, int len);
int write(int fd, char *buf ,int len);
```

参数 fd 用于指定读写操作的套接字描述符；函数 read 的参数 buf 指定接收数据缓冲区，函数 write 的参数 buf 指定发送数据缓冲区；参数 len 指定接收或发送的数据量大小。函数 read 成功执行时，返回读到的数据量大小，否则，返回-1；函数 write 成功执行时，返回写入的数据量大小，否则，返回-1。

8. 套接字地址获取函数

当需要获取套接字的地址时，可以使用 getsockname 函数和 getpeername 函数，前者用于返回本地的套接字地址，后者用于返回与本地套接字建立了连接的对等套接字地址，对其标准调用格式说明如下：

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

getsockname 函数用于获取由描述符 socket 给出的套接字的本地捆绑名，其存储 socket 的地址位于 add 参数所指的 sockaddr 结构对象中，存储其地址长度位于 addrlen 所指对象中。如果地址的实际长度大于 addr 所指对象的长度，则存储的地址将被截断，如果 socket 还没有捆绑地址，则存储在 addr 所指对象中的值将是未定义的。若该函数调用成功，则返回 0，若调用失败则返回 1。

存储在 addr 参数所指对象中的地址格式依赖于该套接字的通信域。对于给定的通信域，套接字地址的长度通常是固定的，如果用户需要确切知道空间大小，并提供实际需要的存储空间，通常的做法是利用与套接字通信域相匹配的数据类型为 addr 所指对象分配空间，然后强制其地址转换为“struct socket *”并传送给 getsockname。

getsockname 函数通常会应用于以下情况：

- 对于没有使用 bind 捆绑地址至套接字的客户进程，在它成功调用 connect 之后，

getsockname 可以返回内核指定给该套接字的本地地址（如 IP 地址和端口号等）。

- 当利用 0 端口号（告诉内核选择本地端口号）调用 bind 之后，getsockname 可以返回内核指定给该套接字的本地端口号。
- getsockname 可以获得一个套接字的地址族。
- 服务进程在接收了客户的连接之后（成功调用 accept 之后），以 accept 返回的描述符调用 getsockname 可以获得指定给该连接的套接字地址，这个套接字是实际连接的套接字，而不是侦听套接字。

函数 getpeername 用于获取一个套接字的远程对等套接字的地址，将返回与 socket 连接的套接字地址，并且存储这个地址于 addr 参数所指对象中，存储该地址的长度于 addrlen 所指对象中，如果地址的实际长度大于 addr 提供的长度，则存储的地址将被截断，调用该函数成功时会返回值 0，如果调用该函数失败则返回-1。


注 意

虽然从 accept 的返回参数中也能得到对等套接字的地址，但是当服务程序是由 accept 的进程通过 fork 和 exec 而执行时，由于 accept 返回参数的存储空间位于父进程中，因此经过 exec 后它将不复存在。在这种情况下，getpeername 是唯一能够获得对等套接字地址的方法。

9. 发送和接收函数

除了之前介绍的读写函数 read 和 write 之外，用户还可以使用 recv 和 send 函数来在套接字中实现数据的发送和接收，recv 和 send 函数类似于标准的 read 和 write 函数，但它们只能用于套接字，并且还需要一个另外的参数，此参数用于指明控制套接字特殊传输方式的各种标志，其标准调用格式如下：

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

send 函数用于启动从 socket 指定的套接字传送一条消息到对等的套接字，如果调用成功，则返回实际发送的字节数，如果失败则返回-1。sockfd 参数为套接字描述符，buffer 为待发送数据的缓冲区，len 为数据长度，但是函数发送的实际长度可能小于其指定的长度，flags 用于指定消息的传送类型，当该值为 0 时 send 函数和 write 函数完全相同，或者使用如下两种取值。

- MSG_OOB：send 函数发送的数据成为带外数据，带外数据是流套接字特有的。在流套接字上传送数据时，数据按它们写出的顺序传送。因为接收进程必须依次读套接字上的当前数据，因此，当出现一个紧急情况时，没有办法立即通知接收进程。带外数据正是用于解决这一问题。带外数据在正常的数据流之外发送，其效果相当于越过套接字上所有等待读的数据。当它到达接收进程时，接收进程会收到一个信号，从而进程可以立即处理这个数据。

- MSG_DONTROUTE：不在消息中包含路由信息，通常来说普通应用不会关心相应的信息。


注 意

send 函数的成功返回值仅仅表明其把 buf 的数据发送出去了，并不代表消息已经被正确地接收。

rev 函数用于从已经连接的套接字接收消息，若调用成功，则返回读到 buffer 所指缓冲区中数据的字节长度，如果没有消息可接收并且对等套接字已执行了 shutdown，将返回 0，否则返回-1。其 socket、buf 和 len 参数和 send 函数中的参数完全相同，flags 参数则用于指明接收到消息的类型，如果该参数为 0，则 rev 函数和 read 函数完全相同，或者使用如下三种取值。

- MSG_OOB：读带外数据。
- MSG_PEEK：窥视套接字上的数据而不实际读出它们，即尽管 buffer 所指对象中填入了所请求的数据，但随后的 read 或 recv 将读到相同的数据。
- MSG_WAITALL：请求函数阻塞直至所请求的全部数据都已接收到。


注 意

read 和 write 函数通常用来读写套接字上的普通数据，当需要发送或接收特殊数据，如带外数据时，就必须使用 send 和 recv 函数才能做到。

write 函数、send 函数、read 函数和 recv 函数都是用于 TCP 协议下面向连接的套接字的数据发送和接收，而在 UDP 协议下面向无连接的套接字数据发送和接受则需要使用 sendto 和 recvfrom 函数，对其标准调用格式说明如下：

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,const struct sockaddr *dest_addr, socklen_t
 addrlen);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,struct sockaddr *src_addr, socklen_t *addrlen);
```

sendto 函数用于在 UDP 协议下发送数据，参数 sockfd 为套接字的描述符，buf 为指向数据发送缓冲区的指针，len 表示将要发送的字节数，flags 一般设置为 0，dest_addr 为指向数据发送的套接口地址数据结构的指针，addrlen 指向套接字数据结构的长度，当调用该函数成功的时候返回实际发送的字节数，调用该函数失败时则返回-1。

recvfrom 函数用于在 UDP 协议下接收数据，参数 buf 指向数据接收缓冲区的指针，sockaddr 为指向数据接收的套接字地址结构的指针，其他参数的含义与 sendto 函数相同，当调用该函数成功的时候返回实际接收的字节数，若调用该函数失败则返回-1。

11.4 Linux 的 TCP 编程

TCP 是 TCP/IP 协议族中面向连接的可靠协议，本小节将介绍其工作流程以及在 Linux 中对其

进行编程的方法。

11.4.1 TCP 基础

同其他任何协议栈一样，TCP 向相邻的高层提供服务。因为 TCP 的上一层就是应用层，因此，TCP 数据传输实现了从一个应用程序到另一个应用程序的数据传递。应用程序通过编程调用 TCP 并使用 TCP 服务，提供需要准备发送的数据，用来区分接收数据应用的目的地址和端口号。

通常情况下，应用程序通过打开一个 socket 来使用 TCP 服务，TCP 管理到其他 socket 的数据传递。可以说，通过 IP 的源/目的可以唯一地区分网络中两个设备的连接，通过 socket 的源/目的可以唯一地区分网络中两个应用程序的连接。

TCP 对话通过三次握手来进行初始化。三次握手的目的是使数据段的发送和接收同步，告诉其他主机其一次可接收的数据量，并建立虚连接。

图 11.9 描述了这三次握手的简单过程：

01 初始化主机通过一个同步标志置位的数据段发出会话请求。

02 接收主机通过发回具有以下项目的数据段表示回复：同步标志置位、即将发送数据段的起始字节的顺序号、应答并带有将收到的下一个数据段的字节顺序号。

03 请求主机再回送一个数据段，并带有确认顺序号和确认号。

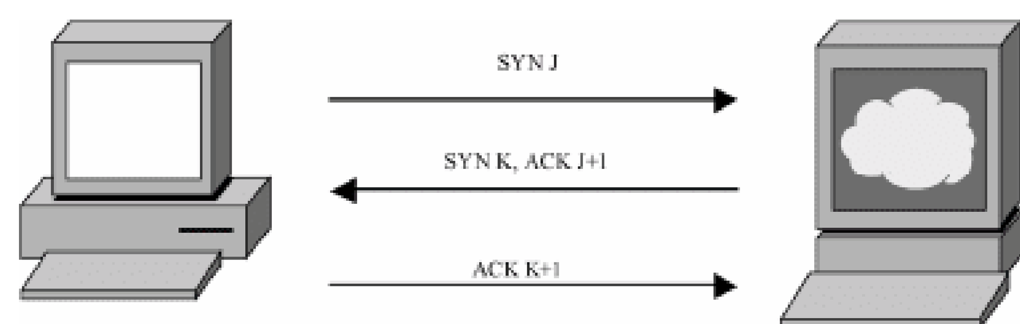


图 11.9 TCP 的三次握手过程示意

TCP 实体所采用的基本协议是滑动窗口协议，当发送方传送一个数据报时，它将启动计时器。当该数据报到达目的地后，接收方的 TCP 实体往回发送一个数据报，其中包含一个确认序号，表示希望收到的下一个数据包的顺序号。如果发送方的定时器在确认信息到达之前超时，那么发送方会重发该数据包。

图 11.10 是 TCP 的数据包头格式，对其各个部分说明如下。

- 源端口、目的端口：16 位长，标识出远端和本地的端口号。
- 序号：32 位长，标识发送数据报的顺序。
- 确认号：32 位长，希望收到的下一个数据包的序列号。
- TCP 头长：4 位长，表明 TCP 头中包含多少个 32 位字。
- 6 位未用。
- ACK：ACK 位置 1 表明确认号是合法的。如果 ACK 为 0，那么数据报不包含确认信息，确认字段被省略。
- PSH：表示是带有 PUSH 标志的数据。接收方只等请求数据包一到便将其送往应用程序，而不必等到缓冲区装满时才传送。
- RST：用于复位由于主机崩溃或其他原因而出现的错误连接，还可以用于拒绝非法的数据包或拒绝连接请求。

- SYN：用于建立连接。
- FIN：用于释放连接。
- 窗口大小：16 位长，窗口大小字段表示在确认了字节之后还可以发送多少个字节。
- 校验和：16 位长，是为了确保高可靠性而设置的，用于校验头部、数据和伪 TCP 头部之和。
- 可选项：0 个或多个 32 位字，包括最大 TCP 载荷、滑动窗口比例以及选择重发数据包等选项。



图 11.10　TCP 的数据包格式

11.4.2　TCP 的工作流程和应用

基于 TCP 传输协议的服务器与客户端间的通信工作流程可以利用如图 11.11 所示的过程来描述。

01 服务器先用 socket 函数来建立一个套接口，用这个套接口完成通信的监听及数据的收发。

02 服务器利用 bind 函数来绑定一个端口号和 IP 地址，使套接口与指定的端口号、IP 地址相关联。

03 服务器调用 listen 函数，使服务器的这个端口和 IP 处于监听状态，等待网络中某一客户机的连接请求。

04 客户机用 socket 函数建立一个套接口，设定远程 IP 和端口。

05 客户机调用 connect 函数连接远程计算机指定的端口。

06 服务器调用 accept 函数来接收远程计算机的连接请求，建立起与客户机之间的通信连接。

07 建立连接以后，客户机利用 write 函数或 send 函数向 socket 中写入数据，也可以使用 read 函数或 recv 函数读取服务器发送来的数据。

08 服务器利用 read 函数或 recv 函数读取客户机发送来的数据，也可以利用 write 函数或 send 函数来发送数据。

09 完成通信以后，使用 close 函数关闭 socket 连接。

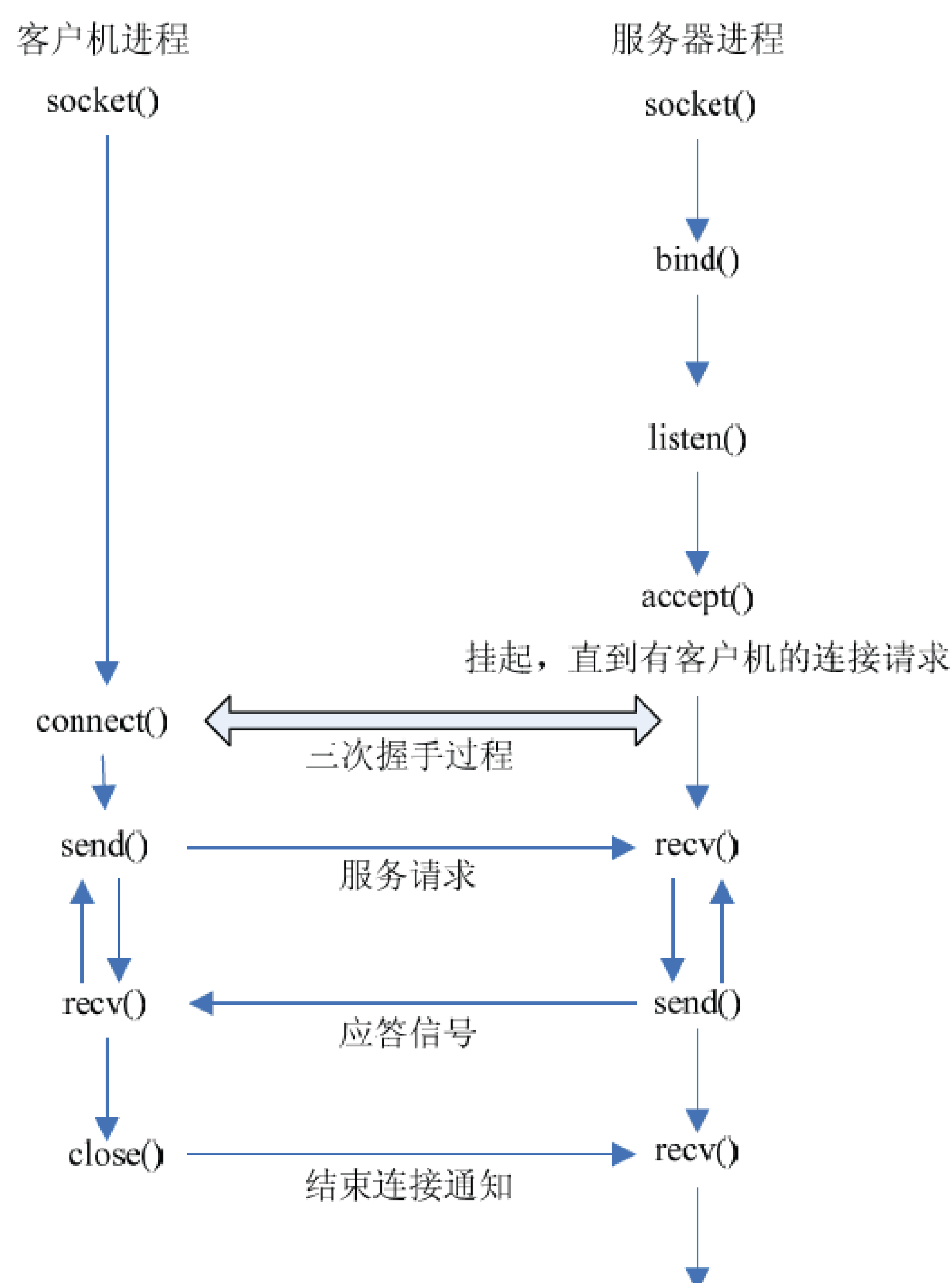


图 11.11　TCP 套接口通信工作流程

例 11.10 和例 11.11 是一个使用 TCP 进行通信的服务器端和客户端应用实例：服务器端接收客户端请求，创建一个子进程来向客户端发送当前系统时间；客户端则读取服务器端发送的信息。

【例 11.10】TCP 通信程序服务器端

应用代码使用端口 25555 作为通信端口，首先调用 socket 函数和 bind 函数建立套接字并且绑定端口，然后调用 listen 函数等待客户端连接，如果有客户端的连接信息，则使用 accept 函数接收该连接并且创建一个子进程，在子进程中发送当前的时间信息，最后在子进程退出的时候关闭该套接字接口。

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <unistd.h>
5   #include <time.h>
6   #include <sys/types.h>
7   #include <netinet/in.h>
8   #include <sys/socket.h>
9
10  #define SERV_PORT 25555                      //服务器接听端口号
11  #define BACKLOG     20                       //请求队列中允许请求数
12  #define BUF_SIZE    256                      //缓冲区大小
```

```
13
14  int main(int argc,char *argv[])
15  {
16      int ret;
17      time_t tt;
18      struct tm *ttm;
19      char buf[BUF_SIZE];
20      pid_t pid;        //定义管道描述符
21      int sockfd;          //定义 sock 描述符
22      int clientfd; //定义数据传输 sock 描述符
23      struct sockaddr_in host_addr;                    //本机 IP 地址和端口信息
24      struct sockaddr_in client_addr;                  //客户端 IP 地址和端口信息
25      int length = sizeof client_addr;
26      //创建套接字
27      sockfd = socket(AF_INET, SOCK_STREAM, 0);        //TCP/IP 协议，数据流套接字
28      if(sockfd == -1)                                 //判断 socket 函数的返回值
29      {
30          printf("创建 socket 失败.\n");
31          return 0;
32      }
33      //绑定套接字
34      bzero(&host_addr, sizeof host_addr);
35      host_addr.sin_family = AF_INET;                  //TCP/IP 协议
36      host_addr.sin_port = htons(SERV_PORT);           //设定端口号
37      host_addr.sin_addr.s_addr = INADDR_ANY;          //本地 IP 地址
38      ret = bind(sockfd, (struct sockaddr *)&host_addr, sizeof host_addr);       //绑定套接字
39      if(ret == -1) //判断 bind 函数的返回值
40      {
41          printf("调用 bind 失败.\n");
42          return 1;
43      }
44      //监听网络端口
45      ret = listen(sockfd, BACKLOG);
46      if(ret == -1)    //判断 listen 函数的返回值
47      {
48          printf("调用 listen 函数失败.\n");
49          return 1;
50      }
51      while(1)
52      {
53          clientfd = accept(sockfd, (struct sockaddr *)&client_addr, &length);     //接收连接请求
54          if(clientfd == -1)
55          {
56              printf("调用 accept 接受连接失败.\n");
57              return 1;
58          }
59          pid = fork();                                //创建子进程
60          if(pid == 0)                                 //在子进程中处理
61          {
```

```
62              while(1)
63              {
64                  bzero(buf, sizeof buf);                      //首先清空缓冲区
65                  tt = time(NULL);
66                  ttm = localtime(&tt);                        //获取当前时间参数
67                  strcpy(buf, asctime(ttm));                   //将时间信息 copy 进缓冲区
68                  send(clientfd, buf, strlen(buf), 0);         //发送数据
69                  sleep(2);
70              }
71              close(clientfd);                                 //调用 close 函数关闭连接
72          }
73          else if(pid > 0)
74          {
75              close(clientfd);                                 //父进程关闭套接字，准备下一个客户端连接
76          }
77      }
78      return 0;
79  }
```

将文件保存为 exam1109TCPSever.c，在终端中使用 gcc 编译链接，生成可执行文件 exam1109TCPSever。

```
alloy@ubuntu:~/linuxc/chapter11$ gcc exam1109TCPSever.c -o exam1109TCPSever
```

【例 11.11】TCP 通信程序客户端

应用代码使用 argv[1]作为连接的 IP 地址，将这个 IP 地址放入 serv_addr 所指定的地址结构体中，在创建套接字之后使用 connect 函数和服务器建立连接，使用 recv 接收服务器发送过来的时间信息并且打印输出。

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <unistd.h>
5   #include <sys/types.h>
6   #include <netinet/in.h>
7   #include <sys/socket.h>
8
9   #define SERV_PORT 25555                                  //服务器接听端口号
10  #define BACKLOG         20                               //请求队列中允许请求数
11  #define BUF_SIZE    256                                  //缓冲区大小
12
13  int main(int argc, char *argv[])
14  {
15      int ret;
16      char buf[BUF_SIZE];
17      int sockfd;                                          //定义 sock 描述符
18      struct sockaddr_in serv_addr;                        //服务器 IP 地址和端口信息
```

```
19     if(argc != 2)
20     {
21         printf("命令行输入有误.\n");                    //命令行带 IP
22         return 1;
23     }
24     //创建套接字
25     sockfd = socket(AF_INET, SOCK_STREAM, 0);          //TCP/IP 协议，数据流套接字
26     if(sockfd == -1)
27     {
28         printf("调用 socket 函数失败.\n");
29         return 2;
30     }
31     //建立连接
32     bzero(&serv_addr, sizeof serv_addr);
33     serv_addr.sin_family = AF_INET;                    //TCP/IP 协议
34     serv_addr.sin_port = htons(SERV_PORT);             //设定端口号
35     //serv_addr.sin_addr.s_addr = INADDR_ANY;          //使用回环地址 127.0.0.1
36     inet_aton(argv[1], (struct sockaddr *)&serv_addr.sin_addr.s_addr);       //设定 IP 地址
37     ret = connect(sockfd, (struct sockaddr *)&serv_addr, sizeof serv_addr);  //绑定套接字
38     if(ret == -1)
39     {
40         printf("调用 connect 函数失败.\n");
41         return 3;
42     }
43     while(1)
44     {
45         bzero(buf, sizeof buf);
46         recv(sockfd, buf, sizeof(buf), 0);             //接收数据
47         printf("接收到: %s", buf);
48         sleep(1);
49     }
50     close(sockfd);    //关闭链接
51     return 0;
52 }
```

将文件保存为 exam1109TCPClient.c，在终端中使用 gcc 编译链接，生成可执行文件 exam1109TCPClient。

在两个不同的终端中分别执行 exam1109TCPClient 和 exam1109TCPSever 可执行文件，可以看到如下的输出。

```
alloy@ubuntu:~/linuxc/chapter11$ ./exam1109TCPClient 192.168.1.5
接收到: Fri Mar 14 09:58:42 2014
接收到: Fri Mar 14 09:58:44 2014
接收到: Fri Mar 14 09:58:46 2014
接收到: Fri Mar 14 09:58:48 2014
接收到: Fri Mar 14 09:58:50 2014
接收到: Fri Mar 14 09:58:52 2014
```

11.5 Linux 的 UDP 编程

UDP 和 TCP 不同，其是一个无连接的协议，所以其建立起来非常简单，不需要进行“三次握手”等操作，但是也具有相对不够安全的缺点。

11.5.1 UDP 的基础知识

UDP 协议从问世至今已经被使用了很多年，虽然其最初的光彩已经被一些类似协议所掩盖，但是在网络质量越来越高的今天，UDP 的应用得到了大大地增强。它比 TCP 协议更为高效，也能更好地解决实时性的问题。如今，包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都使用了 UDP 协议。

UDP 的数据报头如图 11.12 所示，对其各个部分说明如下。

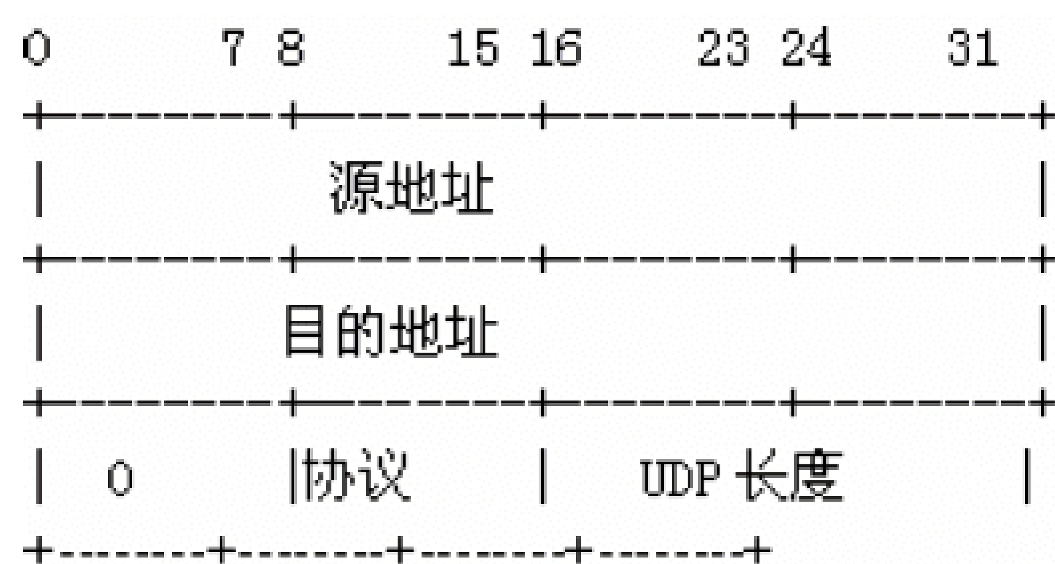


图 11.12　UDP 的数据报头

- 源地址、目的地址：16 位长，用于标识出远端和本地的端口号。
- 数据报的长度：是指包括报头和数据部分在内的总字节数。因为报头的长度是固定的，所以该域主要用来计算可变长度的数据部分（又称为数据负载）。

协议的选择应该考虑到以下 3 个方面。

- 数据可靠性：对数据要求高可靠性的应用需要选择 TCP 协议，如验证、密码字段的传送都是不允许出错的，而对数据的可靠性要求不那么高的应用可选择 UDP 传送。
- 应用实时性：TCP 协议在传送过程中要使用三次握手、重传确认等手段来保证数据传输的可靠性。使用 TCP 协议会有较大的时延，因此不适合对实时性要求较高的应用，如 VoIP、视频监控等。相反，UDP 协议则在这些应用中能发挥很好的作用。
- 网络可靠性：由于 TCP 协议的提出主要是解决网络的可靠性问题，通过各种机制来减少错误发生的概率，因此，在网络状况不是很好的情况下需要选用 TCP 协议（如广域网等情况），但是若在网络状况很好的情况下（如局域网等）就不需要采用 TCP 协议，而建议选择 UDP 协议来减少网络负荷。

11.5.2 UDP 的工作流程和应用

基于 UDP 传输协议的服务器与客户机间的通信工作流程可以利用如图 11.13 所示的过程来描述。将上图与图 11.11 相比较，它们的主要区别在于：使用 TCP 套接口必须先建立连接（如客户进程使用的是 connect 函数，服务器进程使用的是 listen 函数和 accept 函数），而 UDP

套接口不需要预先建立连接，它在调用 socket 函数生成一个套接口后，在服务器端调用 bind 函数绑定一个端口，然后服务器进程挂起 recvfrom 函数调用，等待并接收网络中某一客户机的数据请求，客户端调用 sendto 函数发送数据请求，同样也挂起 recvfrom 函数调用，等待并接收服务器的应答信号。当数据传送完毕后，UDP 套接口中的客户端调用 close 函数释放通信链路，但不再发送“断开连接通知”信息来通知服务器端释放通信链路。

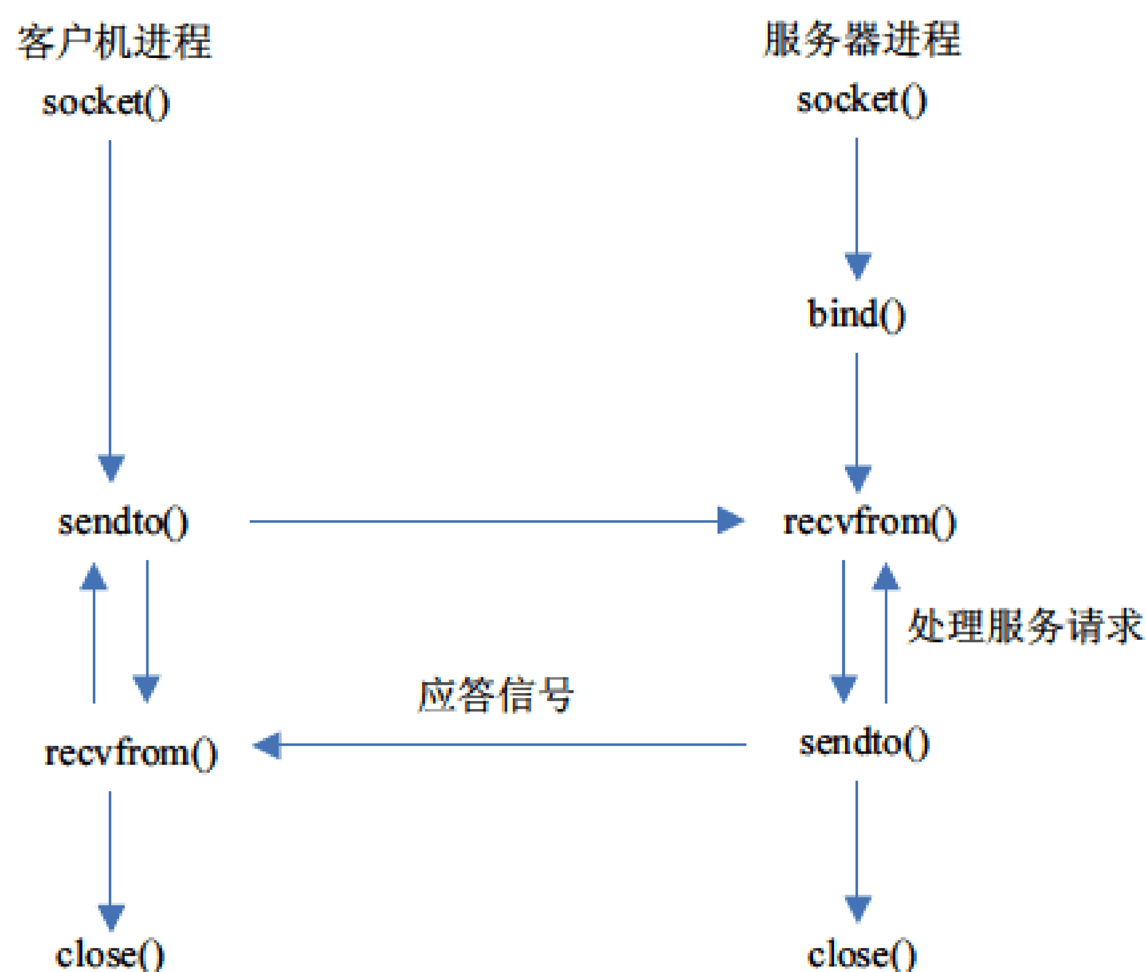


图 11.13　UDP 套接字的工作流程

例 11.12 和例 11.13 是一个使用 UDP 协议进行通信的服务器端和客户端的应用实例，其和 11.4.2 小节中介绍的 TCP 协议实例类似，只是发送时间信息的一方改成了客户端，服务器器端接收客户端发送的时间信息。

【例 11.12】UDP 通信程序服务器端

应用代码使用端口 25555 作为通信端口，首先调用 socket 函数和 bind 函数建立套接字并且绑定端口，然后即可调用 recvfrom 函数来接收数据，将其存放到 buf 缓冲区并且判断其返回值，当 recvfrom 函数的返回值不为-1 时，即可表明接收到客户端的数据，此时调用 printf 函数打印输出 buf 缓冲区的值。

实例的应用代码如下：

```
 1   #include <stdio.h>
 2   #include <stdlib.h>
 3   #include <string.h>
 4   #include <unistd.h>
 5   #include <sys/types.h>
 6   #include <netinet/in.h>
 7   #include <sys/socket.h>
 8
 9   #define SERV_PORT 25555                        //服务器接听端口号
10   #define BACKLOG       20                       //请求队列中允许请求数
11   #define BUF_SIZE    256                        //缓冲区大小
12
13   int main(int argc,char *argv[])
14   {
15       int ret;
```

```
16          char buf[BUF_SIZE];
17          pid_t pid;                                  //定义管道描述符
18          int sockfd;                                 //定义 sock 描述符
19          int clientfd; //定义数据传输 sock 描述符
20          struct sockaddr_in host_addr;               //本机 IP 地址和端口信息
21          struct sockaddr_in client_addr;             //客户端 IP 地址和端口信息
22          int length = sizeof client_addr;
23          //创建套接字
24          sockfd = socket(AF_INET, SOCK_DGRAM, 0);    //TCP/IP 协议，数据流套接字
25          if(sockfd == -1) //判断 socket 函数返回值
26          {
27              printf("创建 socket 失败.\n");
28              return 1;
29          }
30          //绑定套接字
31          bzero(&host_addr, sizeof host_addr);
32          host_addr.sin_family = AF_INET;             //TCP/IP 协议
33          host_addr.sin_port = htons(SERV_PORT);      //设定端口号
34          host_addr.sin_addr.s_addr = INADDR_ANY;     //本地 IP 地址
35          ret = bind(sockfd, (struct sockaddr *)&host_addr, sizeof host_addr);    //绑定套接字
36          if(ret == -1)                                                           //判断 bind 函数返回值
37          {
38              printf("调用 bind 函数失败.\n");
39              return 1;
40          }
41          while(1)
42          {
43              bzero(buf, sizeof buf);
44              ret = recvfrom(sockfd, buf, sizeof(buf), 0 ,(struct sockaddr *)&client_addr, &length);
45              //接收接连请求
46              if(ret == -1)                           //判断 recvfrom 函数的返回值
47              {
48                  printf("接受连接失败");
49                  return 1;
50              }
51          // printf("Client IP:%s \n", inet_ntoa(client_addr.sin_addr.s_addr)); //输出客户端 IP
52              printf("接收到: %s\n", buf);
53              sleep(2);
54          }
55          close(clientfd);                            //关闭连接
56          return 0;
57      }
```

将文件保存为 exam1110UDPSever.c，在终端中使用 gcc 进行编译连接，生成可执行文件 exam1110UDPSever。

```
alloy@ubuntu:~/linuxc/chapter11$ gcc exam1110UDPSever.c -o exam1110UDPSever
```

【例 11.13】UDP 通信程序客户端

应用代码使用第 3 章中介绍的时间相关函数获得了时间信息，并且将其使用 strcpy 函数存放到 buf 缓冲区中，在建立了套接字之后即可调用 sendto 函数来发送 buf 缓冲区的内容。

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <unistd.h>
5   #include <time.h>
6   #include <sys/types.h>
7   #include <netinet/in.h>
8   #include <sys/socket.h>
9
10  #define SERV_PORT 25555                       //服务器接听端口号
11  #define BACKLOG       20                      //请求队列中允许请求数
12  #define BUF_SIZE    256                       //缓冲区大小
13
14  int main(int argc, char *argv[])
15  {
16     int ret;
17     time_t tt;
18     struct tm *ttm;
19     char buf[BUF_SIZE];
20     int sockfd;                                //定义 sock 描述符
21     struct sockaddr_in serv_addr;              //服务器 IP 地址和端口信息
22     if(argc != 2)
23     {
24        printf("命令行输入有误\n");              //命令行带 IP
25        return 1;
26     }
27     /**创建套接字**/
28     sockfd = socket(AF_INET, SOCK_DGRAM, 0);          //TCP/IP 协议，数据流套接字
29     if(sockfd == -1)   //判断 socket 函数的返回值
30     {
31        printf("调用 socket 函数创建链接失败.\n");
32        return 0;
33     }
34     /**建立连接**/
35     bzero(&serv_addr, sizeof serv_addr);
36     serv_addr.sin_family = AF_INET;                   //TCP/IP 协议
37     serv_addr.sin_port = htons(SERV_PORT);            //设定端口号
38     //serv_addr.sin_addr.s_addr = INADDR_ANY;         //使用回环地址  127.0.0.1
39     inet_aton(argv[1], (struct sockaddr *)&serv_addr.sin_addr.s_addr);          //设定 IP 地址
40     while(1)
41     {
42        bzero(buf, sizeof buf);                        //首先清除缓冲区
43        tt = time(NULL);
```

```
44          ttm = localtime(&tt);
45          strcpy(buf, asctime(ttm));                          //复制缓冲区数据
46          sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr *)&serv_addr, sizeof serv_addr);
47          //接收数据，然后放入缓冲区
48          sleep(2);
49      }
50      close(sockfd);
51      return 0;
52  }
```

将文件保存到 exam1110UDPClient.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam1110UDPClient。

```
alloy@ubuntu:~/linuxc/chapter11$ gcc exam1110UDPClient.c -o exam1110UDPClient
```

在两个终端中分别执行客户端和服务器端的可执行文件，可以看到对应的输出。

```
alloy@ubuntu:~/linuxc/chapter11$ ./exam1110UDPClient 192.168.1.5
```

以下是服务器端接收到的时间信息。

```
alloy@ubuntu:~/linuxc/chapter11$ ./exam1110UDPSever
接收到: Fri Mar 14 10:51:00 2014
接收到: Fri Mar 14 10:51:02 2014
接收到: Fri Mar 14 10:51:04 2014
接收到: Fri Mar 14 10:51:06 2014
```

11.6 应用实例——获取网络时间

Network Time Protocol（NTP）协议是用来使计算机时间同步化的一种协议，它可以使计算机对其服务器或时钟源（如石英钟，GPS 等）进行同步化，它可以提供高精确度的时间校正，且可用加密确认的方式来防止恶毒的协议攻击。

NTP 提供准确时间，首先要有准确的时间来源，这一时间应该是国际标准时间 UTC。 NTP 获得 UTC 的时间来源可以是原子钟、天文台、卫星，也可以从 Internet 上获取，这样就有了准确而可靠的时间源。时间是按 NTP 服务器的等级传播，按照距离外部 UTC 源的远近将所有服务器归入不同的 Stratum（层）中。Stratum-1 在顶层，有外部 UTC 接入，而 Stratum-2 则从 Stratum-1 获取时间，Stratum-3 从 Stratum-2 获取时间，以此类推，但 Stratum 层的总数限制在 15 以内。所有这些服务器在逻辑上形成阶梯式的架构，并相互连接，而 Stratum-1 的时间服务器是整个系统的基础。

进行网络协议实现时最重要的是了解协议数据格式。NTP 数据包有 48 个字节，其中 NTP 包头为 16 个字节，时间戳为 32 个字节。其协议格式如图 11.14 所示，对各个部分说明如下。

2	5	8	16	24	32bit
LI	VN	Mode	Stratum	Poll	Precision
Root Delay					
Root Dispersion					
Reference Identifier					
Reference timestamp (64)					
Originate Timestamp (64)					
Receive Timestamp (64)					
Transmit Timestamp (64)					
Key Identifier (optional) (32)					
Message digest (optional) (128)					

图 11.14　NTP 协议的组成

- LI：跳跃指示器，用于警告在当月最后一天的最终时刻插入的迫近闰秒。
- VN：版本号。
- Mode：工作模式。该字段包括以下值：0－预留；1－对称行为；3－客户机；4－服务器；5－广播；6－NTP 控制信息。NTP 协议具有 3 种工作模式，分别为主/被动对称模式、客户/服务器模式、广播模式。在主/被动对称模式中，有一对一的连接，双方均可同步对方或被对方同步，先发出申请建立连接的一方工作在主动模式下，另一方工作在被动模式下；客户/服务器模式与主/被动模式基本相同，唯一的区别在于客户方可被服务器同步，但服务器不能被客户同步；在广播模式中，有一对多的连接，服务器不论客户工作在何种模式下，都会主动发出时间信息，客户根据此信息调整自己的时间。
- Stratum：对本地时钟级别的整体识别。
- Poll：由符号整数表示连续信息间的最大间隔。
- Precision：由符号整数表示本地时钟的精确度。
- Root Delay：表示到达主参考源的一次往复的总延迟，它是包括 15～16 位小数部分的符号定点小数。
- Root Dispersion：表示一次到达主参考源的标准误差，它是包括 15～16 位小数部分的无符号定点小数。
- Reference Identifier：识别特殊参考源。
- Originate Timestamp：这是向服务器请求分离客户机的时间，采用 64 位时标格式。
- Receive Timestamp：这是向服务器请求到达客户机的时间，采用 64 位时标格式。
- Transmit Timestamp：这是向客户机答复分离服务器的时间，采用 64 位时标格式。
- Authenticator（Optional）：当实现了 NTP 认证模式时，主要标识符和信息数字域就包括已定义的信息认证代码（MAC）信息。

例 11.14 是一个利用 NTP 协议来获取当前网络时间的实例，由于 NTP 协议中涉及比较多的时间相关操作，在本应用中仅要求实现 NTP 协议客户端部分的网络通信模块，也就是构造 NTP 协议字段进行发送和接收，最后与时间相关的操作不需要进行处理。NTP 协议是作为 OSI 参考模型的高层协议，比较适合采用 UDP 传输协议进行数据传输，专用端口号为 123。在本实验中，以国家授时中心服务器（IP 地址为 202.72.145.44）作为 NTP（网络时间）服务器，流程如图 11.15 所示。

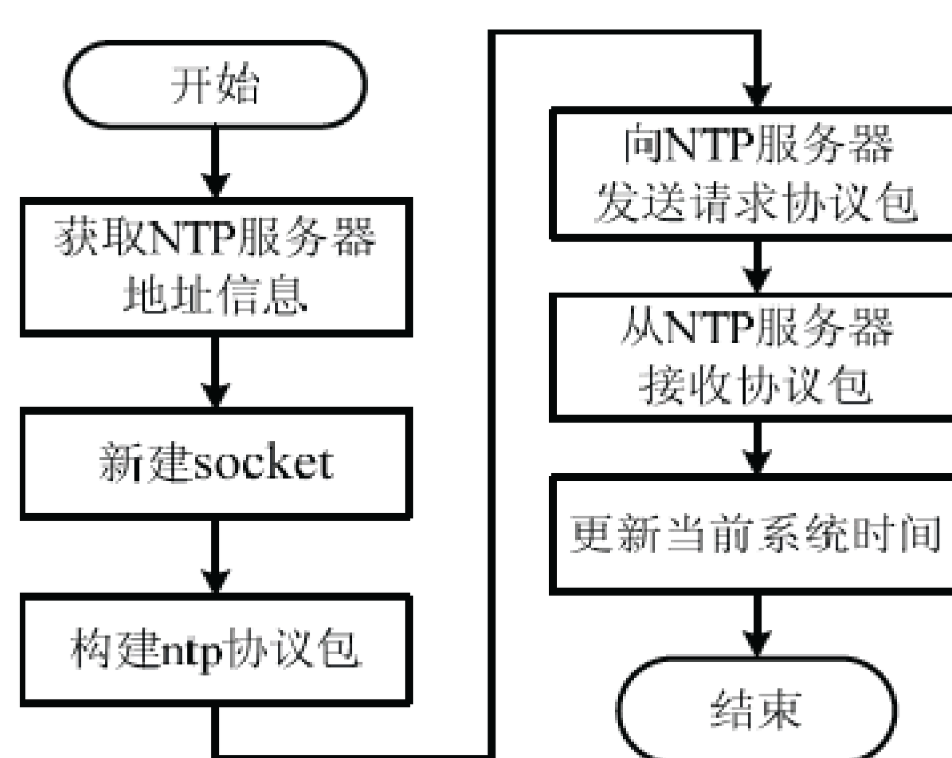


图 11.15 获取网络时间

【例 11.14】获取网络时间

应用代码构建了如下函数用于对网络时间进行操作。

- construct_packet：构建 NTP 协议包。
- get_ntp_time：获取 NTP 时间。
- set_local_time：修改本地时间。

应用代码首先建立和目标主机的连接，然后调用 get_ntp_time 获取 NTP 的时间，然后调用 set_local_time 来修改本地时间。

实例的应用代码如下：

```
1   #include <sys/socket.h>
2   #include <sys/wait.h>
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <errno.h>
6   #include <string.h>
7   #include <sys/un.h>
8   #include <sys/time.h>
9   #include <sys/ioctl.h>
10  #include <unistd.h>
11  #include <netinet/in.h>
12  #include <string.h>
13  #include <netdb.h>
14
15  #define NTP_PORT            123                 /*NTP 专用端口号字符串*/
16  #define TIME_PORT           37                  /* TIME/UDP 端口号 */
17  #define NTP_SERVER_IP       "61.135.250.78"     /*国家授时中心 IP*/
18  #define NTP_PORT_STR        "123"               /*NTP 专用端口号字符串*/
19  #define NTPV1               "NTP/V1"            /*协议及其版本号*/
20  #define NTPV2               "NTP/V2"
21  #define NTPV3               "NTP/V3"
22  #define NTPV4               "NTP/V4"
23  #define TIME                "TIME/UDP"
24
```

```
25   #define NTP_PCK_LEN 48
26   #define LI 0
27   #define VN 3
28   #define MODE 3
29   #define STRATUM 0
30   #define POLL 4
31   #define PREC -6
32
33   #define JAN_1970    0x83aa7e80              /* 1900 年～1970 年之间的时间秒数 */
34   #define NTPFRAC(x)     (4294 * (x) + ((1981 * (x)) >> 11))
35   #define USEC(x)          (((x) >> 12) - 759 * ((((x) >> 10) + 32768) >> 16))
36
37   typedef struct _ntp_time
38   {
39        unsigned int coarse;
40        unsigned int fine;
41   } ntp_time;
42
43   struct ntp_packet
44   {
45          unsigned char leap_ver_mode;
46          unsigned char startum;
47          char poll;
48          char precision;
49          int    root_delay;
50          int    root_dispersion;
51          int reference_identifier;
52          ntp_time reference_timestamp;
53          ntp_time originage_timestamp;
54          ntp_time receive_timestamp;
55          ntp_time transmit_timestamp;
56   };
57
58   char protocol[32];
59   /*构建 NTP 协议包*/
60   int construct_packet(char *packet)
61   {
62          char version = 1;
63          long tmp_wrd;
64          int port;
65          time_t timer;
66          strcpy(protocol, NTPV3);
67          /*判断协议版本*/
68          if(!strcmp(protocol, NTPV1)||!strcmp(protocol, NTPV2)
69                 ||!strcmp(protocol, NTPV3)||!strcmp(protocol, NTPV4))
70          {
71                 memset(packet, 0, NTP_PCK_LEN);
72                 port = NTP_PORT;
73                 /*设置 16 个字节的包头*/
```

```
74              version = protocol[6] - 0x30;
75              tmp_wrd = htonl((LI << 30)|(version << 27)
76                      |(MODE << 24)|(STRATUM << 16)|(POLL << 8)|(PREC & 0xff));
77              memcpy(packet, &tmp_wrd, sizeof(tmp_wrd));
78
79              /*设置 Root Delay、Root Dispersion 和 Reference Indentifier */
80              tmp_wrd = htonl(1<<16);
81              memcpy(&packet[4], &tmp_wrd, sizeof(tmp_wrd));
82              memcpy(&packet[8], &tmp_wrd, sizeof(tmp_wrd));
83              /*设置 Timestamp 部分*/
84              time(&timer);
85              /*设置 Transmit Timestamp coarse*/
86              tmp_wrd = htonl(JAN_1970 + (long)timer);
87              memcpy(&packet[40], &tmp_wrd, sizeof(tmp_wrd));
88              /*设置 Transmit Timestamp fine*/
89              tmp_wrd = htonl((long)NTPFRAC(timer));
90              memcpy(&packet[44], &tmp_wrd, sizeof(tmp_wrd));
91              return NTP_PCK_LEN;
92          }
93          else if (!strcmp(protocol, TIME))/* "TIME/UDP" */
94          {
95              port = TIME_PORT;
96              memset(packet, 0, 4);
97              return 4;
98          }
99          return 0;
100     }
101
102     /*获取 NTP 时间*/
103     int get_ntp_time(int sk, struct addrinfo *addr, struct ntp_packet *ret_time)
104     {
105         fd_set pending_data;
106         struct timeval block_time;
107         char data[NTP_PCK_LEN * 8];
108         int    packet_len, data_len = addr->ai_addrlen, count = 0, result, i, re;
109
110         if (!(packet_len = construct_packet(data)))
111         {
112             return 0;
113         }
114         /*客户端给服务器端发送 NTP 协议数据包*/
115         if ((result = sendto(sk, data,
116                         packet_len, 0, addr->ai_addr, data_len)) < 0)
117         {
118             perror("sendto");
119             return 0;
120         }
121
122         /*调用 select()函数，并设定超时时间为 1s*/
```

```
123         FD_ZERO(&pending_data);
124         FD_SET(sk, &pending_data);
125         block_time.tv_sec=10;
126         block_time.tv_usec=0;
127         if (select(sk + 1, &pending_data, NULL, NULL, &block_time) > 0)
128         {
129             /*接收服务器端的信息*/
130             if ((count = recvfrom(sk, data,
131                         NTP_PCK_LEN * 8, 0, addr->ai_addr, &data_len)) < 0)
132             {
133                 perror("recvfrom");
134                 return 0;
135             }
136 
137             if (protocol == TIME)
138             {
139                 memcpy(&ret_time->transmit_timestamp, data, 4);
140                 return 1;
141             }
142             else if (count < NTP_PCK_LEN)
143             {
144                 return 0;
145             }
146             /* 设置接收 NTP 包的数据结构 */
147             ret_time->leap_ver_mode = ntohl(data[0]);
148             ret_time->startum = ntohl(data[1]);
149             ret_time->poll = ntohl(data[2]);
150             ret_time->precision = ntohl(data[3]);
151             ret_time->root_delay = ntohl(*(int*)&(data[4]));
152             ret_time->root_dispersion = ntohl(*(int*)&(data[8]));
153             ret_time->reference_identifier = ntohl(*(int*)&(data[12]));
154             ret_time->reference_timestamp.coarse = ntohl (*(int*)&(data[16]));
155             ret_time->reference_timestamp.fine = ntohl(*(int*)&(data[20]));
156             ret_time->originage_timestamp.coarse = ntohl(*(int*)&(data[24]));
157             ret_time->originage_timestamp.fine = ntohl(*(int*)&(data[28]));
158             ret_time->receive_timestamp.coarse = ntohl(*(int*)&(data[32]));
159             ret_time->receive_timestamp.fine = ntohl(*(int*)&(data[36]));
160             ret_time->transmit_timestamp.coarse = ntohl(*(int*)&(data[40]));
161             ret_time->transmit_timestamp.fine = ntohl(*(int*)&(data[44]));
162             return 1;
163         } /* end of if select */
164         return 0;
165 }
166 
167 /* 修改本地时间 */
168 int set_local_time(struct ntp_packet * pnew_time_packet)
169 {
170         struct timeval tv;
171         tv.tv_sec = pnew_time_packet->transmit_timestamp.coarse - JAN_1970;
```

```
172         tv.tv_usec = USEC(pnew_time_packet->transmit_timestamp.fine);
173         return settimeofday(&tv, NULL);
174     }
175
176     int main()
177     {
178         int sockfd, rc;
179         struct addrinfo hints, *res = NULL;
180         struct ntp_packet new_time_packet;
181
182         memset(&hints, 0, sizeof(hints));
183         hints.ai_family = AF_UNSPEC;
184         hints.ai_socktype = SOCK_DGRAM;
185         hints.ai_protocol = IPPROTO_UDP;
186         /*调用 getaddrinfo()函数，获取地址信息*/
187         rc = getaddrinfo(NTP_SERVER_IP, NTP_PORT_STR, &hints, &res);
188         if (rc != 0)
189         {
190             perror("getaddrinfo");
191             return 1;
192         }
193         /* 创建套接字 */
194         sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
195         if (sockfd <0 )
196         {
197             perror("socket");
198             return 1;
199         }
200         /*调用取得 NTP 时间的函数*/
201         if (get_ntp_time(sockfd, res, &new_time_packet))
202         {
203             /*调整本地时间*/
204             if (!set_local_time(&new_time_packet))
205             {
206                 printf("NTP client success!\n");
207             }
208         }
209         close(sockfd);
210         return 0;
211     }
```

将文件保存为 exam1111NTC.c，在终端中使用 gcc 进行编译链接，生成可执行文件 exam1111NTC，运行后可以看到获取当前的网络时间成功。

```
alloy@ubuntu:~/linuxc/chapter11$ gcc exam1111NTC.c -o exam1111NTC
alloy@ubuntu:~/linuxc/chapter11$ ./exam1111NTC
NTP client success!
```

11.7 本章习题

1. 编写一个程序，输出当前系统的信息（包括 CPU、操作系统和版本）以及使用的网络字节顺序。

2. 编写一个程序，获取本机的 IP 地址并且将其转换为网络地址。

3. 编写一个程序，使用 socket()函数创建一个 TCP 套接口，并返回该套接字的描述符。

4. 编写一个程序，使用 socket()函数创建一个 UDP 套接口，并返回该套接字的描述符。

5. 编写一对服务器-客户端的应用程序，客户端使用 TCP 向服务器端发送请求日期和时间，服务器端在收到请求后，回答请求并显示出客户的地址。

6. 使用 UDP 协议实现上一题的功能。

第 12 章　在 Linux 中进行基础图形编程

虽然绝大部分的 Linux 操作都可以在命令行下完成，但随着计算机技术的发展，以图形方式显示的计算机操作用户界面 GUI（Graphical User Interface）逐步成为了主流，本章将介绍如何在 Linux 中进行基础图形编程，以一个最简单的 GTK+窗口的实现方法为基础，逐步加上其他功能的实现，涉及以下内容：

- 与 Linux 图形编程相关的基础知识，Qt 和 GTK+的简单介绍。
- GTK+的基础使用方法，包括安装方法、常见数据类型、函数前缀介绍、窗口的创立等。
- 在 GTK+中使用如按钮、组合盒、箭头、输入框等简单构件的方法。
- 在 GTK+中使用组合框、对话框、文件选择等复合构件的方法。
- 在 GTK+中使用菜单和工具栏的方法。

12.1　Linux 图形编程基础

计算机图形界面（GUI）的出现大大方便了用户的操作，其可以使用窗口、菜单、构件（如按钮、滚动条等）来和计算机进行交互，GUI 的组成部分包括桌面、视窗、单一文件界面（Single Document Interface）、多文件界面（Multiple Document Interface）、标签、菜单、图表、按钮等。

在第 1 章的第 1.7.1 小节中对 Linux 的图形界面进行了简单介绍，Linux 提供了多种 GUI 工具包/库以供开发人员使用，它们是用于构造图形界面的集合，其中最常用的是 QT 和 GTK+。

1. QT

QT 是 KDE 图形界面的基础，是一个跨平台的图形用户界面开发库，它不仅支持 Linux 操作系统，还支持所有类型的 Unix 及 Windows 操作系统。

QT 类似于 Windows 平台上的 MFC，是一个 C++工具包，它由几百个 C++类构成，程序员在程序中可以使用这些类。因为 C++是面向对象的编程（Object-Oriented Programming，OOP）语言，而 QT 是基于 C++构造的，所以，QT 也具有 OOP 的所有优点。

2. GTK+

GTK+是 GNOME 图形界面的基础，其采用了纯 C 语言，是开放源代码并且完全免费的（QT 是非完全免费的）。

GTK+（GIMP ToolKit）原本是用于开发 GIMP（一个 GNU 图像处理程序）的工具包，后来逐步被应用于各种图形编程项目，包括 GNOME 图形界面。GTK 是在 GDK（GIMP Drawing Kit）和 gdk-pixbuf 的基础上发展起来的，前者是对访问窗口的底层函数（如 Xlib）的封装，而后者是一个用于客户端图像处理的库。GTK 是一种图形用户界面（GUI）工具包，其本质是一个库（若干

个密切相关的库的集合），其支持创建基于 GUI 的应用程序。用户可以把 GTK+理解为一个工具包，在其包中可以找到用来创建 GUI 的许多已经准备好的构造块。

GTK+提供了包括 C、C++、Perl、Python 等在内的多种语言接口，本章仅介绍其提供的 C 语言相关接口。

图 12.1 是一个 GTK+的层次结构示意，对其各个部分说明如下。

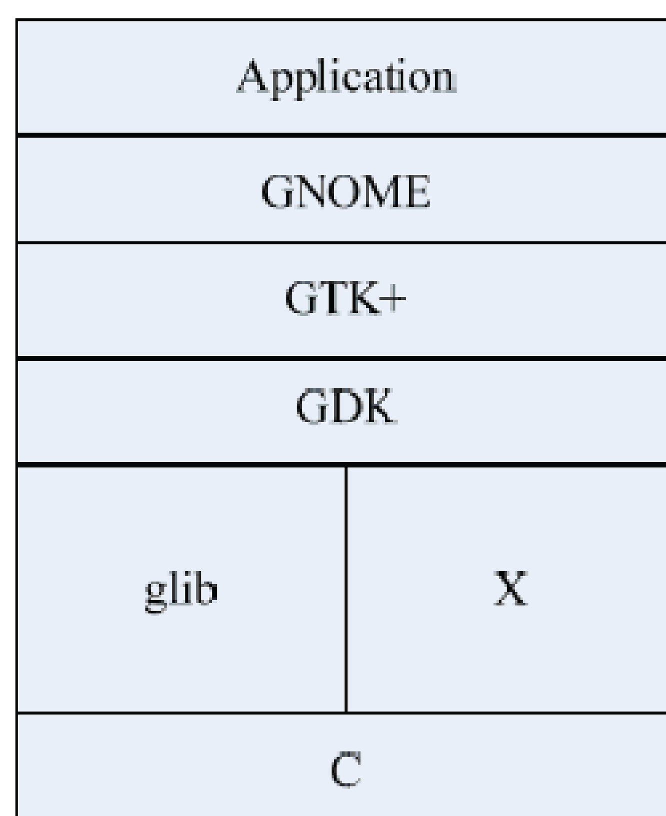


图 12.1 GTK 的层次结构

- C 语言库：在该层中提供了标准 C 的库函数（例如 printf 等）以及 Linux 的系统调用函数（例如 open、fork）以供用户使用。
- glib 库：其包含内存分配、字符串操作、日期和时间、定时器等库函数，也包括链表、队列、树等与数据结构相关的工具函数。
- X 库：其是控制图形显示的底层函数库，包括所有的窗口显示函数、响应鼠标和键盘操作的函数。
- GDK 库：又被称为 GIMP 绘图包，其是为了简化程序员使用 X 函数库而开发的，其对 X 库进行了包装，可以大大提高开发者的工作效率。
- GTK+库：即为 GIMP 工具包，其把 GDK 提供的函数组织成对象，并且使用 C 语言来模拟面向对象的特征，这使得用它开发出来的图形界面程序更为简单和高效。构件（Widget）是 GTK+库的最重要组成部分，包括按钮、标签、文本框等。
- GNOME 库：其是 GTK+库的扩展，与函数、构件进行交互以控制整个桌面。
- Application：具体的应用程序，其负责完成窗口的初始化，创建并显示窗口，进入消息循环，等待用户使用鼠标或键盘进行操作。

在 Linux 下安装 GTK+可以采取从 GTK+ Project 的官方网站（http://www.gtk.org/）下载对应的压缩包，然后编译运行的方法，也可以使用包管理的方法直接从对应的源安装，下面将介绍在 Ubuntu 12.04 下使用 apt-get 包管理工具安装 GTK+的方法，对其详细步骤说明如下：

01 参考第 2.4.1 小节的介绍方法安装好 gcc 编译环境。

02 安装 libgtk2.0-dev 和 libglib2.0-dev 等开发相关的库文件。

```
$sudo apt-get install gnome-core-devel
```

03 安装头文件和库管理工具，用于在编译 GTK+程序时自动找出头文件及库文件位置：

```
$sudo apt-get install pkg-config
```

04 安装 devhelp GTK 文档查看程序：

```
$sudo apt-get install devhelp
```

05 安装 gtk/glib 的 API 参考手册及其他帮助文档：

```
$sudo apt-get install libglib2.0-doc libgtk2.0-doc
```

06 安装 GTK+的图形界面，开发 C/C++语言库：

```
$sudo apt-get install glade libglade2-dev
```

07 安装 gtk 2.0 或者将 gtk+ 2.0 所需的所有文件统一下载安装：

```
$sudo apt-get install libgtk2.0-dev
```

或：

```
$sudo apt-get install libgtk2.0*
//此处的“*”号为通配符，使用该符号可以将与 gtk+ 2.0 相关的所有文件全部安装
```

为了检查当前 Linux 环境下是否已经将 GTK+安装完成，可以使用如下命令：

```
$pkg-config --list-all grep gtk | more
```

该命令可以列举出当前 Linux 环境下已经安装完成的所有 GTK+相关软件包，由于软件包比较多，所以使用了 more 命令来进行分屏显示，其输出如下，可以使用空格翻页：

```
sqlite3                          SQLite - SQL database engine
libmutter                        libmutter - Mutter window manager library
cogl-pango-2.0-experimental      Cogl - An object oriented GL/GLES Abstraction/Utility Layer
gtk-doc                          gtk-doc - API documentation generator
……  //此处省略部分内容
glib-sharp-2.0                   GLib - GLib
gtk+-x11-3.0                     GTK+ - GTK+ Graphical UI Library
--更多--
```

12.2 GTK+的基本使用方法

本节将介绍 GTK+的一些基本使用方法，包括常见数据类型、常见函数前缀、一个最简单的 GTK+窗口、常见基础函数、构件和容器、回调函数等。

12.2.1 GTK+的常见数据类型

GTK+提供了一些常见的数据类型，其实这些类型都是包装好的 C 语言数据类型，其对应关系如表 12.1 所示。

表 12.1 GTK+的常见数据类型

GTK+的数据类型	C 语言数据类型
gchar	char
gint	int
glong	long
gboolean	char
gfloat	float
gdouble	double
guchar	unsigned char
guint	unsigned int
gulong	unsigned long
gpointer	void *
gint8	在任何平台上都是 8 位的整型
gint16	在任何平台上都是 16 位的整型
gint32	在任何平台上都是 32 位的整型
guint8	在任何平台上都是 8 位的无符号整型
guint16	在任何平台上都是 16 位的无符号整型
guint32	在任何平台上都是 32 位的无符号整型

12.2.2 GTK+的常见函数前缀

GTK+提供了一系列函数以供用户调用，这些函数通常会按照实际意义使用一系列的函数前缀，这些前缀及其对应的意义说明如表 12.2 所示。

表 12.2 GTK+的常见函数前缀

函数前缀值	说明
G	glib 定义的数据结构
g	glib 声明的数据类型
g_	glib 定义的函数
gtk_	GTK+定义的函数
Gtk	GTK+库的对象或数据结构
GTK	GTK+定义的宏或者常量

12.2.3 一个最简单的 GTK+窗口

Linux 图形界面下的一个最简单的 GTK+窗口如图 12.2 所示，其在窗口标题栏中显示一个字符串“test”，窗口下方是空白，其他更为复杂的 GTK+窗口都是在这个最简单的窗口基础上增加其他部分得到的，在下一小节中的例 12.1 中将具体介绍这个窗口的实现方法。

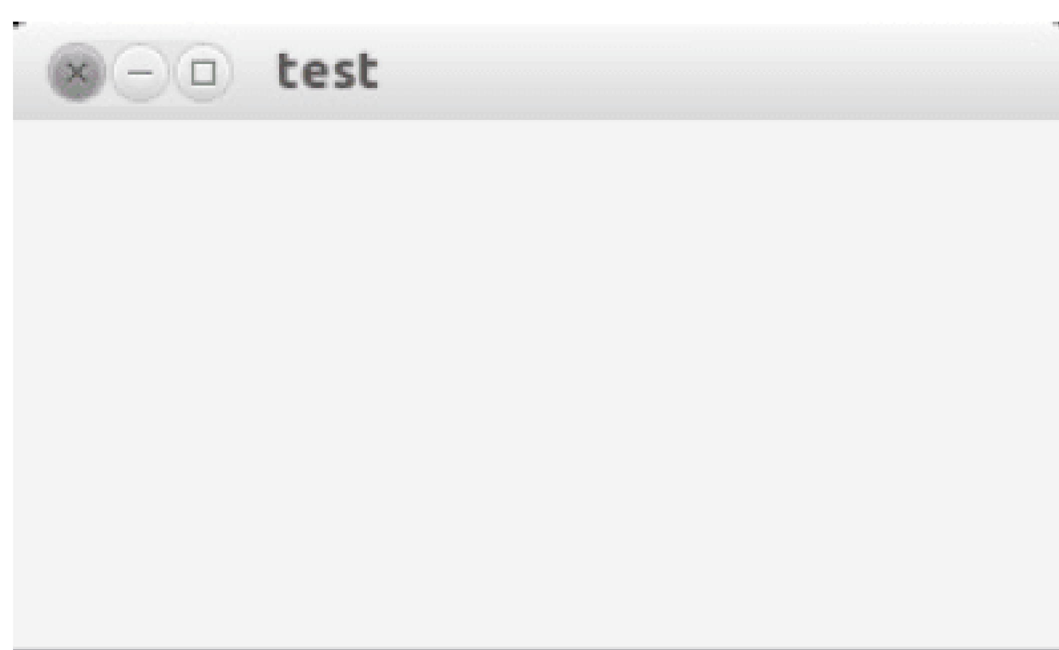


图 12.2　一个简单的 GTK+窗口

12.2.4　GTK+的常见基础函数

本小节将介绍几个与 GTK+相关的基础函数，几乎所有的 GTK+窗口相关代码都会使用到这些函数。

1. gtk_init 函数

gtk_init 函数通常用于对 GTK 的应用代码进行初始化操作，对其标准调用格式说明如下：

```
#include<gtk/gtk.h>
void gtk_init(gint *argc, gchar ***argv );
```

其中 argc 为指向主函数 argc 的指针，argv 为指向主函数 argv 的指针，函数没有返回值，如果在初始化的过程中发生错误，应用程序会立刻退出。

所有的 GTK 应用程序都必须调用该函数，其用于初始化 GTK 所需要的相关库，其在 argv 参数中搜索能识别的运行库参数，主要包括如下一些命令行参数。

- gtk-module：启动时加载指定模块。
- g-fatal-warnings GTK+/GTK：生成的警告和错误会使出错的程序退出。
- gtk-debug：打开指定的 GTK+跟踪/调试消息。
- gtk-no-debug：关闭指定的 GTK+跟踪/调试消息。
- gdk-debug：打开 GDK 中的调试消息。
- gdk-no-debug：关闭指定的 GDK 跟踪/调试消息。
- display h:s:d：连接到指定的 X 服务器，其中 h 为主机名，s 为服务器号，d 为显示号。
- sync：成功建立 X 服务器的连接后，调用 XSynchronize。
- no-xshm：禁止使用 X 共享内存扩展（X Shared Memory Extension）。
- name progname：将程序名设为 progname。
- class classname：程序类是首字母大写的程序名。如果指定了“class”，那么会将程序名设为 classname。

2. gtk_init_check 函数

gtk_init_check 函数用于对 GTK+的应用代码进行初始化操作，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
gboolean gtk_init_check(int *argc,char ***argv);
```

其中 argc 为指向主函数 argc 的指针，argv 为指向主函数 argv 的指针，函数执行成功后返回 TRUE，若出错则返回 FALSE，其功能和 gtk_init 完全相同，区别在于其可以用于判断初始化是否成功的返回值。

GTK+的构件是 GUI 的组成部分，窗口、检查框、按钮和编辑字段都属于构件。通常情况下，将构件和窗口定义为指向 GtkWidget 结构的指针。在 GTK+中，GtkWidget 是用于所有构件和窗口的通用数据类型。

GTK+库进行初始化后，大多数应用建立一个主窗口，在 GTK+中主窗口常常被称为顶层窗口，其不被包含在任何其他窗口内，所以没有上层窗口。在 GTK+中，构件具有父子关系，其中父构件是容器，而子构件是包含在容器中的构件。顶层窗口没有父窗口，但可能成为其他构件的容器。

在 GTK+中建立窗口和构件可以分为如下两步：

01 使用 gtk_window_new 等函数来建立一个窗口和构件。

02 使用 gtk_widget_show 函数来使得构件可见。

3. gtk_window_new 函数

gtk_window_new 函数用于创建一个新的窗口，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_window_new(GtkWindowType type);
```

gtk_window_new 函数的参数 type 用于设置新建窗口的状态和位置，默认状态为一个 200×200 像素大小的窗口，通常来说 type 的取值有如下两种。

- GTK_WINDOW_TOPLEVEL：表示该窗口是一个正常的窗口，窗口可以最小化，当最小化以后，在窗口管理器（相当于 Windows 下的任务栏）中可以看到这一窗口的按钮。
- GTK_WINDOW_POPUP：表示这个窗口是一个弹出式窗口，不可以最小化，但这个窗口是一个独立运行的程序，并不是一个对话框。

如果该函数创建窗体成功，则返回一个 GtkWidget 类型的指针，如果失败则返回一个空指针。对 GtkWidget 类型指针的内部结构说明如下：

```
typedef struct
{
    GtkStyle *style;                        //元件的风格
    GtkRequisition requisition;             //元件的位置
    GtkAllocation allocation;               //元件允许使用的空间
    GtkWindow *window;                      //元件所在的窗口或父窗口
    GtkWidget *parent;                      //元件的父窗口*/
} GtkWidget;
```

4. gtk_window_set_resizable 函数和 gtk_window_get_resizable 函数

GTK 应用程序的窗体大小应该是可变的，这个属性可以由 gtk_window_set_resizable 函数和 gtk_window_get_resizable 函数来设置或获得，对这两个函数的标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_window_set_resizable(GtkWindow *window,gboolean resizable);
gboolean gtk_window_get_resizable(GtkWindow *window);
```

gtk_window_set_resizable 函数用于设置窗体是否大小可变，window 参数为待设置的窗体指针，resizable 为是否可变设置，有 TRUE 和 FALSE 两个可选项，函数没有返回值 gtk_window_get_resizable，用于测试窗体是否大小可变，函数的返回值是一个 gboolean 类型的值，如果窗体大小可变，则返回 TRUE，否则返回 FALSE。

如果希望把窗体设置为不可变，则应该使用如下的语句：

```
gtk_window_set_resizable(GtkWindow (window),FALSE);
```

5. gtk_window_show 函数

gtk_window_show 函数用于将创建好的窗口显示出来，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_widget_show( GtkWidget *widget);
```

其参数 widget 是一个 GtkWidget 类型的结构体，函数没有返回值。

6. gtk_window_set_title 函数

gtk_window_set_title 函数用于设置窗口的标题，对其标准调用格式说明如下。

```
#include <gtk/gtk.h>
void gtk_window_set_title(GTK_WINDOW *window,gchar *title);
```

其中参数 window 为待设置标题的窗口名称，title 为设置的标题字符串（通常来说使用英文，否则在图形界面下可能出现乱码），函数没有返回值。

7. gtk_widget_set_usize 函数和 gtk_widget_set_uposition 函数

gtk_widget_set_usize 函数用于设置窗口的大小，gtk_widget_set_uposition 函数用于设置窗口的位置，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_widget_set_usize(GtkWidget * widget,int x,int y);
void gtk_widget_set_uposition(GtkWidget * widget,int x,int y);
```

其中 widget 为 GtkWidget 类型的结构体，x、y 为当前图形界面坐标系下的坐标值。

8. gtk_main 函数和 gtk_main_quit 函数

gtk_main 函数是 GTK+应用程序的主函数，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_main();
void gtk_main_quit();
```

当 GTK+的应用程序调用 gtk_main 函数之后，即接管了程序的控制权，gtk_main 函数运行主

循环直到 gtk_main_quit 函数被调用，函数没有返回值。

gtk_main_quit 函数用于使 gtk_main 函数跳出循环并且返回。


注 意

GTK+允许 gtk_main 函数进行嵌套操作，对于每一个 gtk_main 函数必须调用一个对应的 gtk_main_quit 函数。

【例 12.1】实现最简单的 GTK+窗口

在第 12.2.3 小节的最后展示了一个 Linux 图形环境下的最简单的 GTK+窗口（图 12.2），这个窗口是所有 GTK+窗口的基础，例 12.1 是使用以上介绍的各个函数来实现该窗口的过程，这个实例也是本章中后续实例的基础，可以看到后面的大部分实例都是在本实例代码的基础上添加对应的控件使用方法完成的。

应用代码使用一个 GtkWidget 类型的指针 window 来描述新创建的窗口，使用字符串数组 title 来存放用于标题的字符串“test”，依次初始化 GTK+、新建窗口、设置窗口标题、设置窗体大小、设置窗体起始坐标，最后将窗体显示出来。

实例的应用代码如下：

```
1   #include <gtk/gtk.h>
2   int main(int argc,char *argv[])
3   {
4           GtkWidget *window;                                      //一个 GtkWidget 类型的结构体指针
5           char title[]="test";
6           gtk_init(&argc, &argv);                                 //初始化 GTK
7           window = gtk_window_new(GTK_WINDOW_TOPLEVEL);           //新建一个 window 窗体
8           gtk_window_set_title(GTK_WINDOW(window), title);
9           //设置窗体的标题为 title 指向的字符串
10            gtk_widget_set_usize(GTK_WINDOW (window),300,150);      //设置窗体的大小
11            gtk_widget_set_uposition(GTK_WINDOW(window),200,200);  //设置窗体的起始坐标位置
12            gtk_widget_show(window);                               //显示窗体
13            gtk_main();                                            //进入主函数
14            return 0;
15  }
```

将文件保存为 exam1201gtksimpletest.c，在终端中编译生成可执行文件 exam1201gtksimpletest，需要注意的是必须使用编译参数`pkg-config --cflags --libs gtk+-2.0`，其中“`”为键盘左边的点号而不是单引号。

```
alloy@ubuntu:~/linuxc/chapter12$ gcc exam1201gtksimpletest.c -o exam1201gtksimpletest `pkg-config
 --cflags --libs gtk+-2.0`
alloy@ubuntu:~/linuxc/chapter12$ ./exam1201gtksimpletest
```

单击运行，可以看到如图 12.2 所示的窗口，但是需要注意的是该窗口无法关闭，即使是单击了窗口上方的“×”，也必须在 Shell 中使用 Ctrl+C 组合键来退出，这是因为在代码中 gtk_main 函数没有退出的关系，在下一小节中将介绍应该如何正确地退出当前窗口。


注 意

由于 GTK+实例的输出都是图形，所以在本章后续实例中不再赘叙述编译和输出过程，会以图片的形式给出它们的运行结果。

12.2.5　GTK+的构件和容器

在第 12.2.4 小节中介绍了窗口的实现方法，但是对于一个完整的 GTK 应用程序来说只有窗口是不够的，其中必须有一些类似按钮、对话框的东西，这些东西被称为 GTK 的构件（GtkWidget），而添加这些构件的载体被称为容器。

1. GTK+的构件及其基础操作函数

GTK 中的最常用构件包括按钮、录入字段、列表框和复合框，所有建立控件的函数都返回一个指向 GtkWidget 的指针，该指针可以调用对控件进行操作的通用函数。


注 意

通常来说在实际使用中还需要通过一些宏定义将这个 GtkWidget 的指针转换为控件对应的专用指针，以供下一步操作，这些宏定义包括：GTK_WIDGET(winget)、GTK_OBJECT(object)、GTK_SIGNAL_FUNC(function)、GTK_CONTAINER(window)、GTK_BOX(box)。

在 GTK 中添加一个构件需要通过如下 5 个步骤：

01 调用函数建立一个构件，并且取得其对应的 GtkWidget 指针。
02 建立该构件的回调函数，以便用户和应用程序进行交互，需要注意的是并不是每个构件都有回调函数或者需要回调函数的。
03 设置该构件的自身属性，包括大小、位置、颜色等。
04 调用函数将该构件添加到容器中（该构件是顶层窗口除外）。
05 调用函数显示该构件。

GTK+的构件有一些基础操作函数，对这些函数介绍如下。

- gtk_widget_set_sensitive：用于改变构件的敏感性，也就是构件的可用性，不敏感的构件通常会以灰色显示，其参数 widget 为需要改变的构件对应的指针，setting 为设置值，有 TRUE 和 FALSE 两个取值，该函数没有返回值，对标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_widget_set_sensitive(GtkWidget *widget,gboolean setting);
```

- gtk_widget_destroy 函数：删除一个构件，其参数 widget 为需要删除的构件对应的指针，该函数没有返回值，对标准调用格式说明如下：

```
#include <gtk/gtk.h>
void    gtk_widget_destroy(GtkWidget *widget);
```

- gtk_widget_hide 函数：用于隐藏一个构件，其参数 widget 为需要隐藏的构件对应的指针，该函数没有返回值。其实被隐藏的构件还是存在的，可以调用 gtk_widget_show 函数来重新显示，对标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_widget_hide (GtkWidget *widget);
```

- gtk_widget_set_size_request 和 gtk_widget_get_size_request 函数：用于设置和获得构件的大小，其参数 widget 为待设置/获得大小的构件指针，width 为构件的宽度，而 height 为构件的高度，单位均为像素，函数没有返回值，对标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_widget_set_size_request (GtkWidget *widget, gint *width, gint *height);
void gtk_widget_get_size_request (GtkWidget *widget, gint *width, gint *height);
```

2. GTK+的容器及其基础操作函数

容器（GtkContainer）是构件的载体，构件需要放在容器中才能被显现出来，最常见的容器是顶层窗口，但是事实上包括按钮在内的许多构件也可以作为容器，需要注意的是一个容器只能容纳一个构件，如果需要容纳多个构件，则需要使用组合框或者组合表。

容器也有一些基础操作函数，对这些函数介绍如下。

- gtk_container_add 函数：向容器中添加一个构件，其中参数 container 为待添加构件的容器名称，widget 为待添加的构件名称，函数没有返回值，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_container_add(GtkContainer *container, GtkWidget *widget);
```

- gtk_container_remove 函数：从容器中移除一个构件，其中参数 container 为待移除构件的容器名称，widget 为待移除的构件名称，函数没有返回值，需要注意的是移除的构件并不会消失，还可以再次添加到容器。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_container_remove(GtkContainer *container, GtkWidget *widget);
```

- gtk_container_set_border_width 和 gtk_container_get_border_width 函数：设置/获得容器的边缘大小，其中参数 container 为待设置/获得边缘大小的容器，参数 border_width 为容器的边缘大小，gtk_container_set_border_width 函数没有返回值，gtk_container_get_border_width 函数的返回值为容器的边缘大小。

```
#include <gtk/gtk.h>
void gtk_container_set_border_width (GtkContainer *container, guint border_width);
guint gtk_container_get_border_width (GtkContainer *container);
```

12.2.6 GTK+的回调函数

GTK 的回调函数是用于对 GTK 应用程序中的信号进行处理的函数，这个信号通常来说是用户

对于构件的各种动作，例如按键被按下等，每个构件都可以登记自己的回调函数，而一个回调函数可以对应多个构件，回调函数需要使用 g_signal_connect 函数来进行注册，类似第 8 章的 singal 信号，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
gulong g_signal_connect(gpointer *object,const gchar *name,GCallback func,gpointer data);
```

其中参数 object 为产生信号的构件，参数 name 为信号的名称，参数 func 为回调函数的名称，参数 data 为传递给回调函数的数据，如果函数成功执行，则返回 TRUE，否则返回 FALSE。

【例 12.2】实现能退出的 GTK+窗口

例 12.1 实现的窗口并不能退出，只能用 Ctrl+C 组合键来退出，可以利用回调函数实现窗口的正常退出，其应用代码如例 12.2 所示，可以看到与例 12.11 相比其只多了一句代码：

```
g_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
```

应用代码中产生信号的构件为 window，产生的信号为 destroy，处理信号的回调函数为 gtk_main_quit，这个函数用于 gtk_main 主函数的退出。

实例的应用代码如下：

```
1   #include <gtk/gtk.h>
2   int main(int argc,char *argv[])
3   {
4           GtkWidget *window;                              //一个 GtkWidget 类型的结构体指针
5           char title[]="test";
6           gtk_init(&argc, &argv);                         //初始化 GTK
7           window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
8           //新建一个 window 窗体
9          gtk_window_set_title(GTK_WINDOW(window), title);
10          //设置窗体的标题为 title 指向的字符串
11          gtk_widget_set_usize(GTK_WINDOW (window),300,150);
12          //设置窗体的大小
13          gtk_widget_set_uposition(GTK_WINDOW(window),200,200);
14          //设置窗体的起始坐标位置
15          g_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
16          //退出的回调函数
17          gtk_widget_show(window);                        //显示窗体
18          gtk_main();                                     //进入主函数
19          return 0;
20          }
```

运行该代码，可以看到在单击关闭“×”按钮之后窗口会自动关闭退出，这是因为此时已经调用了 gtk_main_quit 函数。

【例 12.3】使用用户自编辑的回调函数

用户也可以自行编写一个函数，用于处理窗体的退出，其好处是可以在退出的时候进行更多的操作，例如提示即将退出等。

应用代码定义了一个 gint 类型的 closewindow 函数用于在接收到 destroy 关闭窗体信号之后关闭窗体，在其中调用了 g_print 函数在终端命令行中输出对应的提示信息，然后调用 gtk_main_quit 函数退出窗体，同样在主函数中使用了 g_signal_connect 函数来注册回调函数 closewindow。

实例的应用代码如下：

```
1  #include <gtk/gtk.h>
2  //关闭窗口的回调函数
3  gint closewindow(GtkWidget *widget,gpointer gdata)
4  {
5    g_print("close the window.\n");                    //输出提示
6    gtk_main_quit();                    //调用 gtk_main_quit 来退出主循环，实现关闭窗体
7    return FALSE;                                      //退出
8  }
9  int main(int argc,char *argv[])
10  {
11        GtkWidget *window;                            //一个 GtkWidget 类型的结构体指针
12        char title[]="test";
13        gtk_init(&argc, &argv);                       //初始化 GTK
14        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
15         //新建一个 window 窗体
16        gtk_window_set_title(GTK_WINDOW(window), title);
17        //设置窗体的标题为 title 指向的字符串
18        gtk_widget_set_usize(GTK_WINDOW (window),300,150);
19        //设置窗体的大小
20        gtk_widget_set_uposition(GTK_WINDOW(window),200,200);
21        //设置窗体的起始坐标位置
22        g_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(closewindow),NULL);
23       //注册回调函数
24       gtk_widget_show(window);                       //显示窗体
25       gtk_main();                                    //进入主函数
26       return 0;
27 }
```

代码中涉及的 g_print 函数通常用于调试，可以在终端（shell）输出一个字符串，以观察当前的程序执行状态，其类似于 printf 函数，参数 format 为待输出的字符串，函数没有返回值，对标准调用格式说明如下：

```
#include   <gtk/gtk.h>（也可以#include <glib.h>）
void g_print(gchar *format,…);
```


注 意

对于窗口来说有两个最基础的信号：①destroy：当窗口被关闭的时候发出该信号；②delete_event：即将关闭窗口时发出该信号。

12.3 在 GTK+中使用简单构件

GTK+中的简单构件是相对于组合构件而言的，其通常用于实现某个单一的功能，例如按钮、

标签、输入框等。

12.3.1 按钮

按钮是 GTK+图形界面中最常用的构件，其主要的功能是允许用户单击并且驱动某些操作进行，按钮的事件通常包括按钮的被按下和被释放两种。

GTK+提供了 gtk_button_new 系列函数，用于在窗体中创建一个按钮，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_button_new(void);
GtkWidget *gtk_button_new_with_label(const gchar *label);
```

gtk_button_new 函数用于创建一个不带 label（标签）的按钮，其没有参数，返回值是一个指向 GtkWidget 的指针；gtk_button_new_with_label 函数用于创建一个带 label（标签）的按钮，其参数是一个表示标签内容的字符串，该字符串会显示在按钮上，返回值和 gtk_button_new 相同。

对于按钮而言，其能形成如下事件（即为回调函数可以使用的信号），这些事件也可以由其对应的函数所模拟。

- pressed：按钮被按下，对应 gtk_button_pressed 函数。
- released：按钮被释放，对应 gtk_button_released 函数。
- clicked：单击按钮，即按钮先被按下，然后被释放的整个过程，对应 gtk_button_clicked 函数。
- enter：鼠标移动到按钮上，对应 gtk_button_enter 函数。
- leave：鼠标离开按钮，对应 gtk_button_leave 函数。

【例 12.4】在 GTK+中使用按钮控件

例 12.4 是一个创建按钮的应用实例，应用代码在如图 12.3 所示的窗口中创建一个按钮名称为“Button”的按钮。和窗口类似，当创建按钮之后需要调用 gtk_widget_show 函数来将该按钮显示出来。

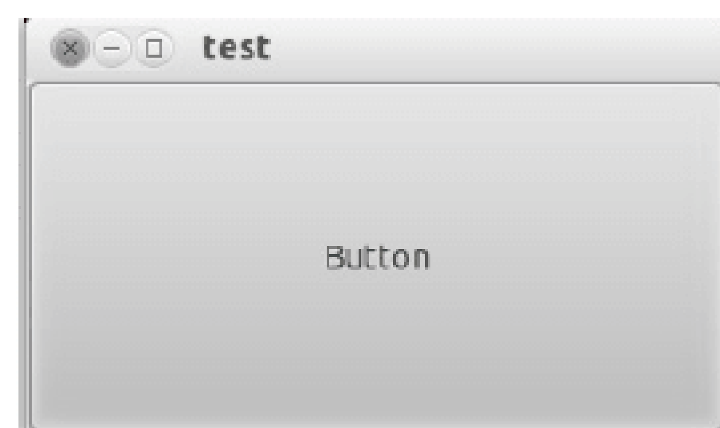


图 12.3　一个名称为“Button”的按钮窗口

应用代码在例 12.3 的基础上添加了按键控件对应的调用代码，在最简单的 GTK+窗口上添加了一个带 label 的名称为 Button 的按钮。

实例的应用代码如下：

```
1    #include <gtk/gtk.h>
2
3    //按键的回调处理函数
```

```
4    void button_deal(GtkWidget *widget,gpointer *data)
5    {
6        g_print("Button event:%s\n",data);                    //输出按键的状态
7    }
8
9    int main(int argc,char *argv[ ])
10   {
11       GtkWidget *window;
12       GtkWidget *button;                                      //窗口和按键的对应指针
13       char title[]="test";
14       gtk_init(&argc,&argv);   //初始化函数
15       window = gtk_window_new(GTK_WINDOW_TOPLEVEL);         //创建窗体
16       gtk_window_set_title(GTK_WINDOW(window), title);     //设置窗体的标题为 title 指向的字符串
17       gtk_widget_set_usize(GTK_WINDOW (window),300,150);          //设置窗体的大小
18       gtk_widget_set_uposition(GTK_WINDOW(window),200,200);   //设置窗体的起始坐标位置
19       gtk_signal_connect(GTK_OBJECT(window),"delete_event",G_CALLBACK(gtk_main_quit),NULL);
20       //登记窗体 delete_event 信号的回调函数
21       button = gtk_button_new_with_label("Button");           //创建带标号的按钮
22       gtk_signal_connect(GTK_OBJECT(button),"pressed",GTK_SIGNAL_FUNC(button_deal),"pressed");
23       //登记按钮 pressed 信号的回调函数
24       gtk_signal_connect(GTK_OBJECT(button),"released",GTK_SIGNAL_FUNC(button_deal),"released");
25       //登记按钮 released 信号的回调函数
26     gtk_signal_connect(GTK_OBJECT(button),"clicked",GTK_SIGNAL_FUNC(button_deal),"clicked");
27       //登记按钮 clicked 信号的回调函数
28       gtk_signal_connect(GTK_OBJECT(button),"enter",GTK_SIGNAL_FUNC(button_deal),"enter");
29       //登记按钮 enter 信号的回调函数
30       gtk_signal_connect(GTK_OBJECT(button),"leave",GTK_SIGNAL_FUNC(button_deal),"leave");
31       //登记按钮 leave 信号的回调函数
32       gtk_container_add(GTK_CONTAINER(window),button);          //把按钮加入窗体
33       gtk_widget_show(button);                                   //显示按钮
34       gtk_widget_show(window);                                   //显示窗体
35       gtk_main();
36       return 0;
37   }
```

编译运行以上代码，对按钮进行单击等操作，可以在 Shell 终端中看到对应的动作提示输出，在实际应用中可以在这些提示输出的位置添加相应的处理函数。

```
Button event:released
Button event:leave
Button event:enter
Button event:leave
```

12.3.2 触发按钮

触发按钮是一种特殊的按钮，其有被按下和释放两种状态，其特点是被按下后并不会自行弹起，而是需要再次点击。

通常来说可以使用 gtk_toggle_button_new 函数和 gtk_toggle_button_new_with_label 函数来建立

一个新的触发按键，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_toggle_button_new(void);
GtkWidget *gtk_toggle_button_new_with_labal(const gchar *label);
```

这两个函数的返回值都是新建的触发按钮，gtk_toggle_button_new_with_label 函数的参数是按钮上的标签字符串。

【例 12.5】在 GTK+中使用触发按钮控件

触发按钮的使用方法和普通按钮基本相同，只是其多了一个触发事件（信号）toggled，例 12.5 是触发按钮的应用实例，在其中可以看到其有"被按下"和"释放弹起"两种不同的状态，其外形也如图 12.3 所示，触发按钮控件的应用代码和按钮基本上完全相同，只是调用的具体函数不同。

实例的应用代码如下：

```
1   #include <gtk/gtk.h>
2
3   //按键的回调处理函数
4   void button_deal(GtkWidget *widget,gpointer *data)
5   {
6     g_print("Button event:%s\n",data);                          //输出按键的状态
7   }
8
9   int main(int argc,char *argv[ ])
10  {
11    GtkWidget *window;
12    GtkWidget *button;                                          //窗口和按键的对应指针
13    char title[]="test";
14    gtk_init(&argc,&argv);                                      //初始化函数
15    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);               //创建窗体
16    gtk_window_set_title(GTK_WINDOW(window), title); //设置窗体的标题为 title 指向的字符串
17    gtk_widget_set_usize(GTK_WINDOW (window),300,150);          //设置窗体的大小
18    gtk_widget_set_uposition(GTK_WINDOW(window),200,200);       //设置窗体的起始坐标位置
19    gtk_signal_connect(GTK_OBJECT(window),"delete_event",G_CALLBACK(gtk_main_quit),NULL);
20    //登记窗体 delete_event 信号的回调函数
21    button = gtk_toggle_button_new_with_label("Button");        //创建带标号的触发按钮
22    gtk_signal_connect(GTK_OBJECT(button),"pressed",GTK_SIGNAL_FUNC(button_deal),"pressed");
23    //登记按钮 pressed 信号的回调函数
24    gtk_signal_connect(GTK_OBJECT(button),"released",GTK_SIGNAL_FUNC(button_deal),"released");
25    //登记按钮 released 信号的回调函数
26    gtk_signal_connect(GTK_OBJECT(button),"clicked",GTK_SIGNAL_FUNC(button_deal),"clicked");
27    //登记按钮 clicked 信号的回调函数
28    gtk_signal_connect(GTK_OBJECT(button),"enter",GTK_SIGNAL_FUNC(button_deal),"enter");
29    //登记按钮 enter 信号的回调函数
30    gtk_signal_connect(GTK_OBJECT(button),"leave",GTK_SIGNAL_FUNC(button_deal),"leave");
31    //登记按钮 leave 信号的回调函数
32    gtk_signal_connect(GTK_OBJECT(button),"toggle",GTK_SIGNAL_FUNC(button_deal),"toggle");
33    //登记按钮 toggle 信号的回调函数
```

```
34        gtk_container_add(GTK_CONTAINER(window),button);    //把按钮加入窗体
35        gtk_widget_show(button);                            //显示按钮
36        gtk_widget_show(window);                            //显示窗体
37        gtk_main();
38        return 0;
39    }
```

12.3.3　复选框

复选框又被称为检查按钮，其和触发按钮的主要差别在于图像（外观）的表现上。

可以使用 gtk_check_button_new 和 gtk_check_button_new_with_label 函数来创建一个复选框，参数 label 为复选框的标签也即为显示的内容，函数的返回值都是指向按钮的指针，对其标准调用格式说明如下。

```
#include <gtk/gtk.h>
GtkWidget *gtk_check_button_new(void);
GtkWidget *gtk_check_button_new_with_labal(const gchar *label);
```

【例 12.6】在 GTK+中使用复选框控件

例 12.6 是一个在窗体中建立如图 12.4 所示的复选框实例，其触发事件和触发按钮完全相同，也是 toggle，可以看到应用实例和例 12.5 相比只是调用的创建函数不同。

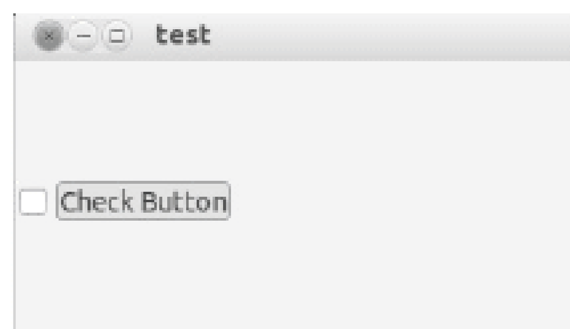


图 12.4　窗口中的复选框

实例的应用代码如下：

```
1     #include <gtk/gtk.h>
2
3     //按键的回调处理函数
4     void button_deal(GtkWidget *widget,gpointer *data)
5     {
6         g_print("Button event:%s\n",data);                        //输出按键的状态
7     }
8
9     int main(int argc,char *argv[ ])
10    {
11        GtkWidget *window;
12        GtkWidget *button;                                        //窗口和按键的对应指针
13        char title[]="test";
14        gtk_init(&argc,&argv);   //初始化函数
15        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);             //创建窗体
16        gtk_window_set_title(GTK_WINDOW(window), title);
          //设置窗体的标题为 title 指向的字符串
17        gtk_widget_set_usize(GTK_WINDOW (window),300,150);        //设置窗体的大小
18        gtk_widget_set_uposition(GTK_WINDOW(window),200,200);     //设置窗体的起始坐标位置
```

```
19        gtk_signal_connect(GTK_OBJECT(window),"delete_event",G_CALLBACK(gtk_main_quit),NULL);
20        //登记窗体 delete_event 信号的回调函数
21        button = gtk_check_button_new_with_label("Check Button"); //创建带标号的触发按钮
22        gtk_signal_connect(GTK_OBJECT(button),"pressed",GTK_SIGNAL_FUNC(button_deal),"pressed");
23        //登记按钮 pressed 信号的回调函数
24        gtk_signal_connect(GTK_OBJECT(button),"released",GTK_SIGNAL_FUNC(button_deal),"released");
25        //登记按钮 released 信号的回调函数
26        gtk_signal_connect(GTK_OBJECT(button),"clicked",GTK_SIGNAL_FUNC(button_deal),"clicked");
27        //登记按钮 clicked 信号的回调函数
28        gtk_signal_connect(GTK_OBJECT(button),"enter",GTK_SIGNAL_FUNC(button_deal),"enter");
29        //登记按钮 enter 信号的回调函数
30        gtk_signal_connect(GTK_OBJECT(button),"leave",GTK_SIGNAL_FUNC(button_deal),"leave");
31        //登记按钮 leave 信号的回调函数
32        gtk_signal_connect(GTK_OBJECT(button),"toggle",GTK_SIGNAL_FUNC(button_deal),"toggle");
33        //登记按钮 toggle 信号的回调函数
34        gtk_container_add(GTK_CONTAINER(window),button);    //把按钮加入窗体
35        gtk_widget_show(button);                            //显示按钮
36        gtk_widget_show(window);                            //显示窗体
37        gtk_main();
38        return 0;
39    }
```

12.3.4 单选框

和复选框对应的是单选框，又被称为单选按钮，由复选框派生而来，区别在于任何时候同一组按钮只能选择其中一个按钮，当单击选择其中一个按钮的时候会自动释放其他按钮而选中当前按钮。

由于单选框通常都是以组的形式存在的，所以建立单选框的操作应该包括如下三个步骤：

01 建立一个新的单选框。

02 将该单选框加入到一个组中。

03 继续建立新的单选框并且加入组中，直到所有需要的单选框都被添加。

在 GTK+中可以使用 gtk_radio_button_new 函数来创建一个不带标签的单选框，使用 gtk_radio_button_new_with_label 来创建一个带标签的单选框，参数 group 为创建的单选框需要加入的组，参数 label 为带标签的单选框的标签字符串，函数的返回值都是指向新建单选框的指针。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_radio_button_new(GSList *group);
GtkWidget *gtk_radio_button_new_with_labal(GSList*group,const gchar *label);
```

在 GTK+中可以使用函数 gtk_radio_button_group 将新建的单选框添加到当前组中，其中参数 radio 为单选框，函数的返回值为单选框添加的组。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GSList *gtk_group_button_group(GtkWidget *radio);
```

每次添加单选框之后都应该调用 gtk_radio_button_group 以获得当前组的信息，然后再次添加单选框，需要注意的是在第一次使用该组之前应该将组指向 NULL，否则会出现错误。

【例 12.7】在 GTK+中使用单选框控件

实例 12.7 是一个在窗口中建立了多个单选框的实例，如图 12.5 所示，由于在一个容器中只能添加一个构件，所以在其中使用了 box（组合盒）这个概念，box 的使用方法会在第 12.3.5 小节中进行详细介绍。

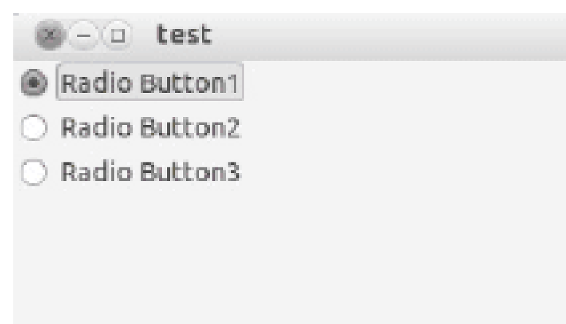


图 12.5　在窗口中建立多个单选框

实例的应用代码如下：

```
1    #include <gtk/gtk.h>
2
3    int main(int argc,char *argv[ ])
4    {
5      GtkWidget *window;
6      GtkWidget *button;                    //窗口和按键的对应指针
7      GtkWidget *box;                       //组装盒
8      GSList *group = NULL;                 //一个用于存放单选框的组
9      char title[]="test";
10     gtk_init(&argc,&argv);                //初始化函数
11     window = gtk_window_new(GTK_WINDOW_TOPLEVEL);      //创建窗体
12     gtk_window_set_title(GTK_WINDOW(window), title);     //设置窗体的标题为 title 指向的字符串
13     gtk_widget_set_usize(GTK_WINDOW (window),300,150);      //设置窗体的大小
14     gtk_widget_set_uposition(GTK_WINDOW(window),200,200);   //设置窗体的起始坐标位置
15     gtk_signal_connect(GTK_OBJECT(window),"delete_event",G_CALLBACK(gtk_main_quit),NULL);
16     //登记窗体 delete_event 信号的回调函数
17     box = gtk_vbox_new(FALSE,0);   //创建一个新的组合盒
18     button = gtk_radio_button_new_with_label(group,"Radio Button1");     //创建单选框
19     group = gtk_radio_button_group(GTK_RADIO_BUTTON(button));      //将单选框添加到组中
20     gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,0);          //将按钮添加到组合盒中
21     gtk_widget_show(button);   //显示按钮
22
23     button = gtk_radio_button_new_with_label(group,"Radio Button2");     //创建单选框
24     group = gtk_radio_button_group(GTK_RADIO_BUTTON(button));      //将单选框添加到组中
25     gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,0);          //将按钮添加到组合盒中
26     gtk_widget_show(button);   //显示按钮
27
28     button = gtk_radio_button_new_with_label(group,"Radio Button3");     //创建单选框
29     group = gtk_radio_button_group(GTK_RADIO_BUTTON(button));      //将单选框添加到组中
30     gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,0);          //将按钮添加到组合盒中
31     gtk_widget_show(button);   //显示按钮
32
```

```
33        gtk_container_add(GTK_CONTAINER(window),box); //把组合盒加入窗体
34        gtk_widget_show(box);       //显示组合盒
35        gtk_widget_show(window); //显示窗体
36        gtk_main();
37        return 0;
38    }
```


注　意

在实例的应用中并没有对单选框的选中操作建立对应的回调函数，即没有对对应事件进行处理，读者可以自行添加。

12.3.5　组合盒

在前面介绍过，一个容器只能存放一个构件，如果想在一个容器中容纳多个构件则需要使用组合盒（box）和组合表（table），后者将在第 12.3.6 小节中进行介绍。

组合盒（GtkBox）可以容纳多个构件，其本身也可以被看作一个构件，但是该构件在运行时并不会显示。组合盒可以分为纵向组合盒和横向组合盒，前者在垂直方式堆积构件，后者在水平方向堆积构件。

GTK+提供了一些基础函数用于对组合盒进行操作，对这些函数说明如下：

- 创建组合盒函数 gtk_hbox_new 和 gtk_vbox_new：这两个函数分别用于创建纵向（gtk_hbox_new）和横向（gtk_vbox_new）的组合盒，函数的参数 homogeneous 用于设定组合盒内的构件是否具有相同的大小，有 FALSE 和 TRUE 两种不同的取值，如果设置为 TRUE，则其中的构件大小都会按照最大的构件来设置，如果设置为 FALSE，则会按照构件的实际大小来设置；参数 spacing 用于设置构件之间的空隙大小，其单位是像素，如果设置为 0，则所有的构件之间会紧密连接在一起；函数的返回值均为指向新创建组合盒的指针。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_hbox_new(gboolean homogeneous,gint spacing);
GtkWidget *gtk_vbox_new(gboolean homogeneous,gint spacing);
```

- gtk_box_pack_start 和 gtk_box_pack_end 函数：这两个函数用于将一个构件加入组合盒中，前者将构件放在组合盒从上到下（纵向组合盒）或者从左到右（横向组合盒）的第一个位置，后者则正好相反。其中参数 box 为指向待放入构件的组合盒的指针，参数 child 为指向待放入构件的指针，参数 expend 用于设置将所有的构件加入组合盒之后构件之间是否还保留可供扩充的空间（如果 homogenous 设置为 TRUE，则可以忽略该参数），fill 参数用于设置是否需要充分利用构件周边的空间，其有 TRUE 和 FALSE 两个取值，前者允许构件略微扩大，后者则禁止；padding 参数用于设置构件周边需要保留的空间像素值，通常设置为 0 即可，这两个函数都没有返回值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_box_pack_start(GtkBox *box,GtkWidget *child,gboolean expend,gboolean fill,guint padding);
void gtk_box_pack_end(GtkBox *box,GtkWidget *child,gboolean expend,gboolean fill,guint padding);
```

【例 12.8】在 GTK+中使用组合盒控件

例 12.8 是在窗口中添加了 4 个竖排按钮的实例，如图 12.6 所示，应用代码分 4 次依次调用了创建按钮的函数 gtk_button_new_with_label，然后使用 gtk_box_pack_start 函数将刚刚创建的按钮添加到组合盒。

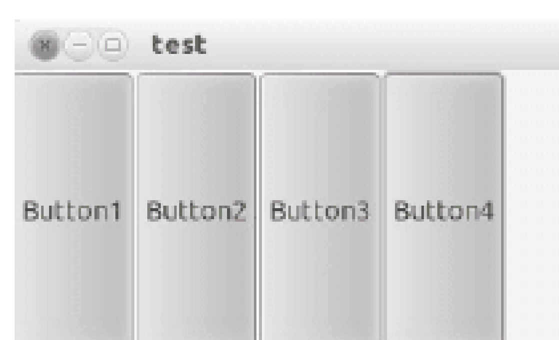


图 12.6　在窗口中添加竖排按钮

实例的应用代码如下：

```
1      #include <gtk/gtk.h>
2
3  int main(int argc,char *argv[ ])
4    {
5      GtkWidget *window;
6      GtkWidget *button;                 //窗口和按键的对应指针
7      GtkWidget *box;                    //组合盒
8      char title[]="test";
9      gtk_init(&argc,&argv);             //初始化函数
10
11     window = gtk_window_new(GTK_WINDOW_TOPLEVEL);       //创建窗体
12     gtk_window_set_title(GTK_WINDOW(window), title);       //设置窗体的标题为 title 指向的字符串
13     gtk_widget_set_usize(GTK_WINDOW (window),300,150);       //设置窗体的大小
14     gtk_widget_set_uposition(GTK_WINDOW(window),200,200);    //设置窗体的起始坐标位置
15     gtk_signal_connect(GTK_OBJECT(window),"delete_event",G_CALLBACK(gtk_main_quit),NULL);
16     //登记窗体 delete_event 信号的回调函数
17     box = gtk_hbox_new(FALSE,0);                            //创建一个新的组合盒，横向
18
19     button = gtk_button_new_with_label("Button1");          //创建一个新的按钮
20     gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,0);   //将按钮添加到组合盒中
21     gtk_widget_show(button);
22
23     button = gtk_button_new_with_label("Button2");          //创建一个新的按钮
24     gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,0);   //将按钮添加到组合盒中
25     gtk_widget_show(button);
26
27     button = gtk_button_new_with_label("Button3");          //创建一个新的按钮
28     gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,0);   //将按钮添加到组合盒中
29     gtk_widget_show(button);
30
31     button = gtk_button_new_with_label("Button4");          //创建一个新的按钮
32     gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,0);   //将按钮添加到组合盒中
33     gtk_widget_show(button);
34
35     gtk_container_add(GTK_CONTAINER(window),box);           //把组合盒加入窗体
```

```
36      gtk_widget_show(box);                                           //显示组合盒
37      gtk_widget_show(window);                                        //显示窗体
38      gtk_main();
39      return 0;
40  }
```

12.3.6　组合表

组合盒只有横向和纵向两种控制方式，如果想对容器内的构件进行更加复杂的控制，可以使用组合表，其允许利用行和列来控制构件的放置，且构件可以取多行或多列。

GTK+同样提供了一系列基础函数用于对组合表进行操作，对这些函数说明如下。

- gtk_table_new 函数：用于创建一个新的组合表，函数的参数 rows 用于指定组合表的行数，cloumns 用于指定表的列数，homogeneous 用于指定构件是否具有相同的大小（和组合盒相同），有 TRUE 和 FALSE 两种不同的选择，函数的返回值为一个指向组合表的指针。组合表的行编号从 0 开始到 rows-1，列编号从 0 开始到 columns-1，每个构件可以占用组合表中需要提供的开始行/列和结束行/列。对其标准调用格式说明如下:

```
#include <gtk/gtk.h>
GtkWidget* gtk_table_new(guint rows,guint columns,gboolean homogeneous);
```

- gtk_table_attach 和 gtk_table_attach_defaults 函数：用于将构件添加到指定的组合表中，其中参数 table 为指向待添加构件的组合表指针，参数 child 为指向待添加构件的指针，参数 left_attach、right_attach、top_attach 和 botton_attach 分别用于表示构件在表中的横向起始位置、横向结束位置、纵向起始位置和纵向结束位置。参数 xoptions 和 yoptions 的取值为 GTK_FILL、GTK_SHRINK 和 GTK_EXPAND 中的一个或者任意组合。其中 GTK_FILL 选项表示构件充分利用分配给它的空间进行扩展；GTK_SHRINK 选项允许构件缩小到比原来分配的空间还小的空间；GTK_EXPAND 选项使表扩展填满它插入的所有空间。参数 xpadding 和 ypadding 用于设置围绕构件填充的像元数。两个函数都没有返回值，其基本功能是一样的，gtk_table_attach_defaults 函数取比较少的参数，并对 gtk_table_attach 函数使用的 xoptions、yoptions、xpadding 和 ypadding 参数用缺省值来代替。对 gtk_table_attach_defaults 函数来说，xpadding 和 ypadding 的缺省值为 0；而 xoptions 和 yoptions 的缺省值为（GTK_FILL|GTK_EXPAND）。对其标准调用格式说明如下:

```
#include <gtk/gtk.h>
void gtk_table_attach(GtkTable *table,GtkWidget * child,guint left_attach,guint right_attach,
guint top_attach,guint    botton_attach, GtkAttachOptions xoptions,GtkAttachOptions yoptions,guint xpadding,
guint ypadding);
void gtk_table_attach_defaults( GtkTable *table,GtkWidget * child,guint left_attach,guint right_attach,
guint top_attach,guint    botton_attach);
```

【例 12.9】在 GTK+中使用组合表控件

例 12.9 是一个在窗口中添加三个按钮的实例，其输出如图 12.7 所示。可以看到其中的按钮 Button3 比其他几个都大，这是因为其使用了函数 gtk_table_attach_defaults 而不是函数

gtk_table_attach，图 12.8 是使用 gtk_table_attach 的输出结果。

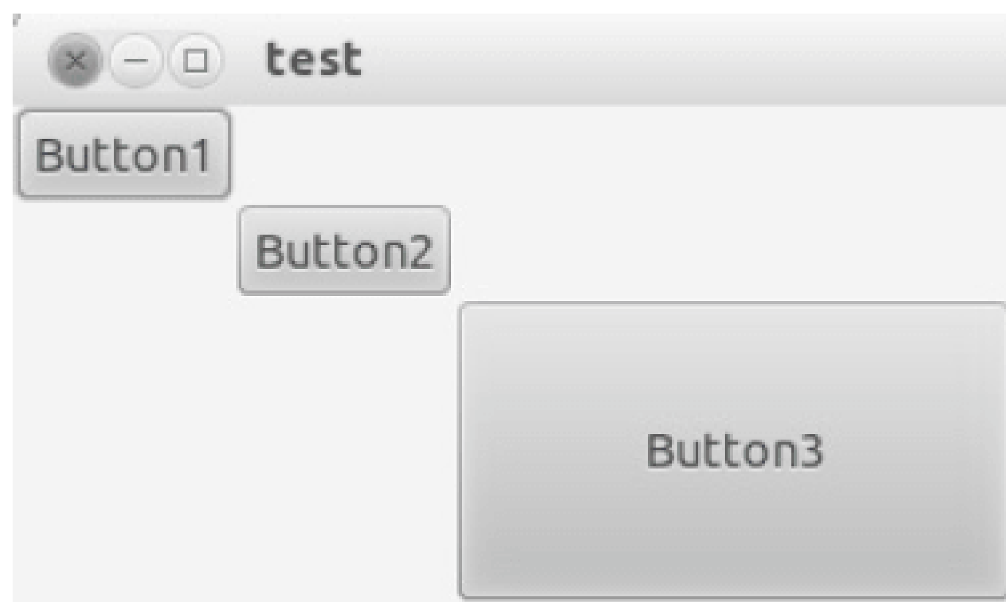


图 12.7　使用组合表在窗体中添加按钮



图 12.8　同样大小的按钮布局

实例的应用代码如下：

```
1       #include <gtk/gtk.h>
2
3       int main(int argc,char *argv[ ])
4       {
5          GtkWidget *window;
6          GtkWidget *button;          //窗口和按键的对应指针
7          GtkWidget *table;           //组合表
8          char title[]="test";
9          gtk_init(&argc,&argv);    //初始化函数
10
11         window = gtk_window_new(GTK_WINDOW_TOPLEVEL); //创建窗体
12         gtk_window_set_title(GTK_WINDOW(window), title);
           //设置窗体的标题为 title 指向的字符串
13         gtk_widget_set_usize(GTK_WINDOW (window),300,150);   //设置窗体的大小
14         gtk_widget_set_uposition(GTK_WINDOW(window),200,200); //设置窗体的起始坐标位置
15         gtk_signal_connect(GTK_OBJECT(window),"delete_event",G_CALLBACK(gtk_main_quit),NULL);
16         //登记窗体 delete_event 信号的回调函数
17
18         table = gtk_table_new(4,4,FALSE);      //创建一个 4 行 4 列的组合表
19
20         button = gtk_button_new_with_label("Button1");    //创建一个按钮
21         gtk_table_attach(GTK_TABLE(table),button,0,1,0,1,GTK_FILL,GTK_FILL,0,0);
           //将按钮添加到组合表
22         gtk_widget_show(button);    //显示按钮
23
24         button = gtk_button_new_with_label("Button2");    //创建一个按钮
25         gtk_table_attach(GTK_TABLE(table),button,1,2,1,2,GTK_FILL,GTK_FILL,0,0);
           //将按钮添加到组合表
26         gtk_widget_show(button);    //显示按钮
27
28         button = gtk_button_new_with_label("Button3");    //创建一个按钮
29         gtk_table_attach_defaults(GTK_TABLE(table),button,2,3,2,3);   //将按钮添加到组合表
30         gtk_widget_show(button);    //显示按钮
31
32         gtk_container_add(GTK_CONTAINER(window),table); //把组合盒加入窗体
33         gtk_widget_show(table);      //显示组合盒
```

```
34        gtk_widget_show(window); //显示窗体
35        gtk_main();
36        return 0;
37    }
```


注 意

除了可以使用组合盒和组合表来指定构件位于容器中的位置之外，GTK 还提供了一个固定的容器构件（GtkFixed）用于将构件放在窗体的固定位置，有兴趣的读者可以自行参考相应的资料。

12.3.7 标签

标签（GtkLabel）是 GTK+中最常用的构件，通常用于显示静态的不可编辑的内容，需要注意的是标签本身是没有信号输出的，如果需要进行交互操作，则需要配合事件盒构件共同使用。

标签的操作函数如下。

- gtk_label_new 函数：用于创建一个新的标签，参数 str 为标签显示的字符串，函数的返回值为指向新创建标签的指针。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget gtk_label_new(char *str);
```

- gtk_label_set_text 函数：用于修改已经创建标签的显示字符串，其中参数 str 为修改后标签显示的字符串，label 为指向待修改字符串内容标签的指针，函数没有返回值。需要说明的是字符串可以含有换行符，GTK 会自动调整字符串内的布局。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_label_set_text(GtkLabel *label,char *str);
```

- gtk_label_get 函数：用于获得标签的显示字符串内容，其中参数 label 为指向待获得显示字符串内容的标签，str 为获得的现实内容，函数没有返回值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_label_get(GtkLabel *label,char **str);
```

- gtk_label_set_justify 函数：用于调整标签正文的对齐方式，其中参数 label 为指向目标标签的指针，jtype 为标签正文的对齐方式，有 GTK_JUSTIFY_LEFT（左对齐）、GTK_JUSTIFY_RIGHT（右对齐）、GTK_JUSTIFY_CENTER（居中对齐）和 GTK_JUSTIFY_FILL（充满）4 种不同的取值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_label_set_justify(GtkLabel *label,GtkJustification jtype);
```

此外可以使用 gtk_label_set_line_wrap 函数来控制标签内容是否自动换行，使用

gtk_label_set_pattern 函数为标签的显示内容添加下划线，读者可以参阅相应的说明手册。

【例 12.10】在 GTK+中使用标签控件

例 12.10 是一个使用单选框和标签综合应用的实例，其根据用户的选择在标签中显示不同的内容，如图 12.9 所示，应用代码通过按钮的回调函数来调用 gtk_label_set 函数，以便修改标签的显示内容。

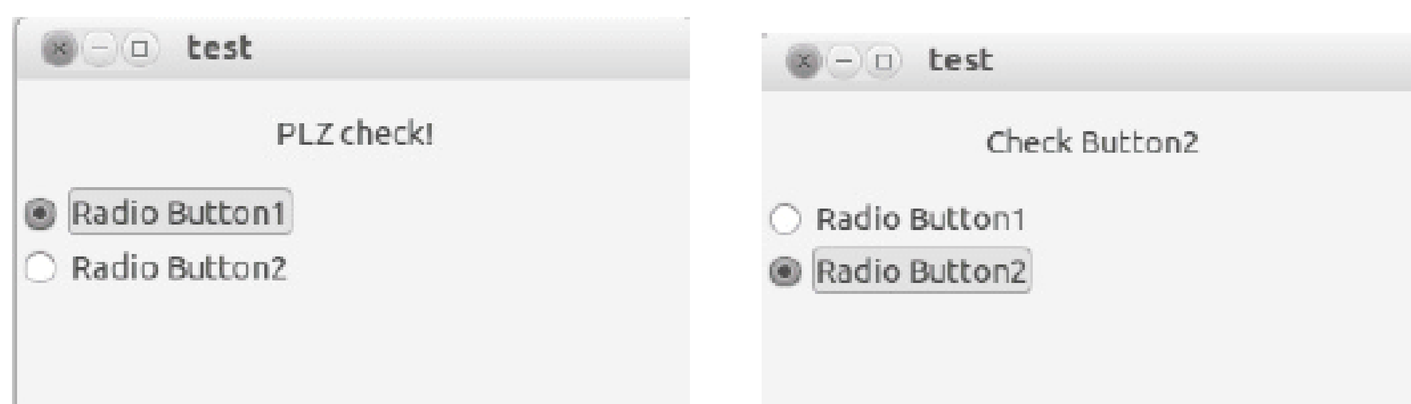


图 12.9　标签和单选框的配合使用

实例的应用代码如下：

```
1   #include <gtk/gtk.h>
2
3   GtkWidget *label;      //标签，由于其需要在回调函数中使用，必须使用全局变量
4
5   //处理按键的回调函数
6   void button_deal(GtkWidget* widget,gpointer* data)
7   {
8       gtk_label_set(GTK_LABEL(label),(char *)data);              //修改标签的内容
9   }
10
11  int main(int argc,char *argv[ ])
12  {
13      GtkWidget *window;
14      GtkWidget *button;                                         //窗口和按键的对应指针
15      GtkWidget *box;                                            //组合盒
16      GSList *group = NULL;                                      //单选框分组
17
18      char title[]="test";
19      gtk_init(&argc,&argv);                                     //初始化函数
20
21      window = gtk_window_new(GTK_WINDOW_TOPLEVEL);        //创建窗体
22      gtk_window_set_title(GTK_WINDOW(window), title);       //设置窗体的标题为 title 指向的字符串
23      gtk_widget_set_usize(GTK_WINDOW (window),300,150);       //设置窗体的大小
24      gtk_widget_set_uposition(GTK_WINDOW(window),200,200);    //设置窗体的起始坐标位置
25      gtk_signal_connect(GTK_OBJECT(window),"delete_event",G_CALLBACK(gtk_main_quit),NULL);
26      //登记窗体 delete_event 信号的回调函数
27
28      box = gtk_vbox_new(FALSE,0);                                       //创建一个新的组合盒,纵向
29
30      label = gtk_label_new("PLZ check!");                               //创建标签
31      gtk_box_pack_start(GTK_BOX(box),label,FALSE,FALSE,15); //将标签添加到组合盒
32
33      button = gtk_radio_button_new_with_label(group,"Radio Button1");      //创建按钮
```

```
34      group = gtk_radio_button_group(GTK_RADIO_BUTTON(button));          //将按钮添加到组中
35      gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,0);             //将按钮加入组合盒
36      gtk_signal_connect(GTK_OBJECT(button),"pressed",GTK_SIGNAL_FUNC(button_deal),"Check
        Button1");
37      //添加按键 1 的 press 事件处理回调函数
38      gtk_widget_show(button);                                           //显示按键
39
40      button = gtk_radio_button_new_with_label(group,"Radio Button2");   //创建按钮
41      group = gtk_radio_button_group(GTK_RADIO_BUTTON(button));          //将按钮添加到组中
42      gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,0);             //将按钮加入组合盒
43      gtk_signal_connect(GTK_OBJECT(button),"pressed",GTK_SIGNAL_FUNC(button_deal),"Check
         Button2");
44      //添加按键 1 的 press 事件处理回调函数
45      gtk_widget_show(button);                                           //显示按键
46
47      gtk_container_add(GTK_CONTAINER(window),box);                      //把组合盒加入窗体
48      gtk_widget_show(label);                                            //显示标签
49      gtk_widget_show(box);                                              //显示组合盒
50      gtk_widget_show(window);                                           //显示窗体
51      gtk_main();
52      return 0;
53    }
```

12.3.8 输入框

如果在 GTK+中需要输入一个字符串，可以使用输入框，这是一个单行输入构件，可以用于输入和显示正文内容。

GTK+提供了一些基础操作函数，以便对输入框进行操作。

- gtk_entry_new 和 gtk_entry_new_with_max_length 函数：这两个函数用于创建一个新的输入框，其中参数 max 用于说明该输入框能接收的最大字符串长度，函数的返回值都是指向新建输入框的指针。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_entry_new(void);
GtkWidget *gtk_entry_new_with_max_length(gint max);
```

- gtk_entry_get_text 函数：该函数用于获得输入框的输入内容，其中参数 entry 为指向需要获得输入内容的输入框指针，函数的返回值是一个指向输入框内容字符串的指针。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
const gchar *gtk_entry_get_text(GtkEntry *entry);
```

- gtk_entry_prepend_text、gtk_entry_append_text 和 gtk_entry_set_text 函数：用于设置输入框的内容，其中参数 text 为待写入输入框的内容，entry 为指向待写入内容的输入框指针，函数没有返回值。gtk_entry_prepend_text 函数用于在已有的字符串开始位置插入新的字

符串，gtk_entry_append_text 函数用于在已有字符串的最后位置插入新的字符串，而 gtk_entry_set_text 函数首先清除原有的所有字符串，然后输入新的字符串。对它们的标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_entry_prepend_text(GtkEntry *entry,const gchar *text);
void gtk_entry_append_text(GtkEntry *entry,const gchar *text);
void gtk_entry_set_text(GtkEntry *entry,const gchar *text);
```

- gtk_entry_set_visibility 函数：用于设置用户是否可以看到输入字段的内容，例如密码等字符串通常都不显示，其中参数 entry 为指向待操作的输入框的指针，visible 参数用于设置输入内容是否可见，其有 TRUE 和 FALSE 两种不同的取值，当设置为 FALSE 时输入框的内容不可见，函数没有返回值。对该函数的标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_entry_set_visibility(GtkEntry *entry,gboolean visible);
```

需要注意的是输入框其实是文本框的简化版本，其和文本框一样都支持 Linux 图形界面的快捷操作组合键。

- Ctrl+A：全选。
- Ctrl+X：剪切到剪切板。
- Ctrl+C：复制到剪切板。
- Ctrl+V：从剪切板粘贴。

【例 12.11】在 GTK+中使用输入框控件实现登录界面

例 12.11 是一个输入框的典型应用实例，其实现了一个登录界面，在如图 12.10 所示的窗口中输入用户名和密码，其中密码采用了不显示输入字符的方式，同时使用了 g_print 函数将用户名和密码在终端中输出显示。



图 12.10　输入框的应用

实例的应用代码如下：

```
1    #include <gtk/gtk.h>
2
3    GtkWidget *name;                       //用户名
4    GtkWidget *passwd;                     //密码
5
```

```
6                                                        //处理按键的回调函数
7   void button_deal(GtkWidget* widget,gpointer* data)
8   {
9      const gchar *uname;
10     const gchar *upasswd;
11
12     uname = (gchar *)malloc(sizeof(gchar));           //分配内存空间
13     upasswd = (gchar *)malloc(sizeof(gchar));         //分配内存空间
14     uname = gtk_entry_get_text(GTK_ENTRY(name));      //获得用户名
15     upasswd = gtk_entry_get_text(GTK_ENTRY(passwd));  //获得密码
16     g_print("Name:%s/n",uname);
17     g_print("Passwd:%s\n",upasswd);
18  }
19
20  int main(int argc,char *argv[ ])
21  {
22     GtkWidget *window;
23     GtkWidget *button;                                //窗口和按键的对应指针
24     GtkWidget *box;                                   //组合盒
25     GtkWidget *label;
26
27     char title[]="test";
28     gtk_init(&argc,&argv);   //初始化函数
29
30     window = gtk_window_new(GTK_WINDOW_TOPLEVEL);       //创建窗体
31     gtk_window_set_title(GTK_WINDOW(window), title);
       //设置窗体的标题为 title 指向的字符串
32     gtk_widget_set_usize(GTK_WINDOW (window),300,150);         //设置窗体的大小
33     gtk_widget_set_uposition(GTK_WINDOW(window),200,200);   //设置窗体的起始坐标位置
34     gtk_signal_connect(GTK_OBJECT(window),"delete_event",G_CALLBACK(gtk_main_quit),NULL);
35     //登记窗体 delete_event 信号的回调函数
36
37     box = gtk_vbox_new(FALSE,0);                               //创建组合框
38     label = gtk_label_new("name:");                            //创建标签
39     gtk_box_pack_start(GTK_BOX(box),label,FALSE,FALSE,5);      //将标签加入组合框
40     gtk_widget_show(label);      //显示标签
41
42     name = gtk_entry_new();                                    //创建输入构件
43     gtk_entry_set_visibility(GTK_ENTRY(name),TRUE);            //设置字符串可见
44     gtk_box_pack_start(GTK_BOX(box),name,FALSE,FALSE,5);       //将输入构件加入组合盒
45     gtk_widget_show(name);                                     //显示输入构件
46
47     label = gtk_label_new("passwd:"); //创建标签
48     gtk_box_pack_start(GTK_BOX(box),label,FALSE,FALSE,5);      //将标签加入组合框
49     gtk_widget_show(label);      //显示标签
50
51     passwd = gtk_entry_new();    //创建输入构件
52     gtk_entry_set_visibility(GTK_ENTRY(passwd),FALSE);         //设置字符串不可见
53     gtk_box_pack_start(GTK_BOX(box),passwd,FALSE,FALSE,5);  //将输入构件加入组合盒
```

```
54        gtk_widget_show(passwd); //显示输入构件
55
56        button = gtk_button_new_with_label("ENTER");                    //创建按钮
57        gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,5);          //将按钮加入组合盒
58        gtk_signal_connect(GTK_OBJECT(button),"pressed",GTK_SIGNAL_FUNC(button_deal),"enter");
          //声明回调函数
59        gtk_widget_show(button);                                        //显示按钮
60
61        gtk_container_add(GTK_CONTAINER(window),box); //把组合盒加入窗体
62        gtk_widget_show(box);                                           //显示组合盒
63        gtk_widget_show(window);                                        //显示窗体
64        gtk_main();
65        return 0;
66    }
```

运行实例，在其中进行输入，可以看到终端中有对应的输出：

```
alloy@ubuntu:~/linuxc/chapter12$ ./exam1211entry
Name:alloy/nPasswd:
Name:alloy/nPasswd:alloy
```

12.3.9　箭头

GTK 的箭头构件（GtkArrow）用于建立一个带有箭头的按钮，其有多种不同的指向方向和不同的风格。需要说明的是箭头构件和标签类似，只是一个指示标识，需要放入一个容器才能使用，本身也不能引发事件，通常来说和按钮配合使用（在箭头构件中放入按钮构件）形成带指示性的按钮。

通常来说可以使用函数 gtk_arrow_new 来创建一个箭头构件，对其标准调用格式说明如下：

```
#include    <gtk/gtk.h>
GtkWidget *gtk_arrow_new(GtkArrowType arrow_type,GtkShadowType shadow_type);
```

其中参数 arrow_type 用于指示箭头的方向，有如下 4 个选择项。

- GTK_ARROW_UP：向上。
- GTK_ARROW_DOWN：向下。
- GTK_ARROW_LEFT：向左。
- GTK_ARROW_RIGHT：向右。

参数 shadow_type 用于指示箭头的投影类型，有 GTK_SHADOW_IN、GTK_SHADOW_OUT、GTK_SHADOW_ETCHED_IN 和 GTK_SHADOW_ETCHED_OUT 共 4 个选择项。

函数的返回值是指向新建的箭头构件指针。

【例 12.12】在 GTK+中使用箭头控件

例 12.12 是一个创建带按钮的左、右指向箭头的实例，其窗口显示如图 12.11 所示，通常来说这个应用可以用于指示滚动条。

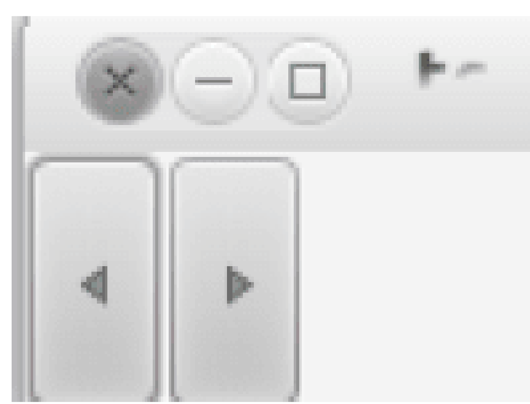


图 12.11　带按钮的左右指向箭头

实例的应用代码如下：

```
1     #include <gtk/gtk.h>
2
3     int main(int argc,char *argv[ ])
4     {
5        GtkWidget *window;
6        GtkWidget *button;                          //窗口和按键的对应指针
7        GtkWidget *arrow;                           //箭头对应的指针

8        GtkWidget *box;                             //组合盒对应的指针
9
10       char title[]="test";
11       gtk_init(&argc,&argv);                      //初始化函数
12       window = gtk_window_new(GTK_WINDOW_TOPLEVEL);      //创建窗体
13       gtk_window_set_title(GTK_WINDOW(window), title); //设置窗体的标题为 title 指向的字符串
14       gtk_widget_set_usize(GTK_WINDOW (window),100,50);       //设置窗体的大小
15       gtk_widget_set_uposition(GTK_WINDOW(window),200,200);   //设置窗体的起始坐标位置
16       gtk_signal_connect(GTK_OBJECT(window),"delete_event",G_CALLBACK(gtk_main_quit),
         NULL);
17       //登记窗体 delete_event 信号的回调函数
18
19       box = gtk_hbox_new(FALSE,0);                            //创建一个组合盒
20       gtk_container_add(GTK_CONTAINER(window),box);           //将组合盒加入窗体
21
22       button = gtk_button_new(); //创建不带标签的按钮
23       arrow = gtk_arrow_new(GTK_ARROW_LEFT,GTK_SHADOW_OUT);    //创建指针
24       gtk_container_add(GTK_CONTAINER(button),arrow);         //将指针加入按钮
25       gtk_box_pack_start(GTK_BOX(box),button,FALSE,TRUE,0);   //将按钮加入组合盒
26       gtk_widget_show(arrow);
27       gtk_widget_show(button);     //显示箭头和按钮
28
29       button = gtk_button_new(); //创建不带标签的按钮
30       arrow = gtk_arrow_new(GTK_ARROW_RIGHT,GTK_SHADOW_OUT);    //创建指针
31       gtk_container_add(GTK_CONTAINER(button),arrow);         //将指针加入按钮
32       gtk_box_pack_start(GTK_BOX(box),button,FALSE,TRUE,0);   //将按钮加入组合盒
33       gtk_widget_show(arrow);
34       gtk_widget_show(button);                                //显示箭头和按钮
35
36       gtk_widget_show(box);                                   //显示按钮
37       gtk_widget_show(window);                                //显示窗体
38       gtk_main();
39       return 0;
```

```
40      }
```

12.3.10 标尺

如果需要在窗口中指示鼠标指针的位置，可以使用标尺构件，每个窗口上都可以放置一个水平标尺构件（GtkHRuler）和一个垂直标尺构件（GtkVRuler）。

在 GTK+中可以使用对应的函数创建标尺或者设置标尺，对这些函数说明如下。

- gtk_hruler_new 和 gtk_vruler_new 函数：分别用于创建水平和垂直标尺，对其标准调用格式说明如下，函数没有参数，返回值为指向标尺的指针。

```
#include <gtk/gtk.h>
GtkWidget *gtk_hruler_new(void);
GtkWidget *gtk_vruler_new(void);
```

- gtk_ruler_set_metric 函数：用于设置标尺的度量单位，其中参数 ruler 为指向待设置度量单位标尺的指针，参数 metric 为度量单位，有 GTK_PIXELS（像素）、GTK_INCHES（英寸）和 GTK_CENTIMETERS（厘米）三种不同的取值，函数没有返回值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_ruler_set_metric(GtkRuler *ruler, GtkMetricType metric);
```

- gtk_ruler_set_range 函数：用于设置标尺的跨度和指示器的初始位置，函数没有返回值，对其中各个参数说明如下：ruler 为指向待操作的标尺指针；lower 为标尺的开始值；upper 为标尺的结束值；position 为标尺的指针指示器的初始位置；max_size 为显示的最大值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_ruler_set_range(GtkRuler *ruler, gfloat lower, gfloat upper, gfloat position, gfloat max_size);
```

【例 12.13】在 GTK+中使用标尺控件

例 12.13 是一个在窗口中添加横向标尺的实例，其输出如图 12.12 所示，由于箭头控件也必须放到组合盒中，所以应用代码在创建标尺之前使用 gtk_vbox_new 函数创建了一个组合盒。

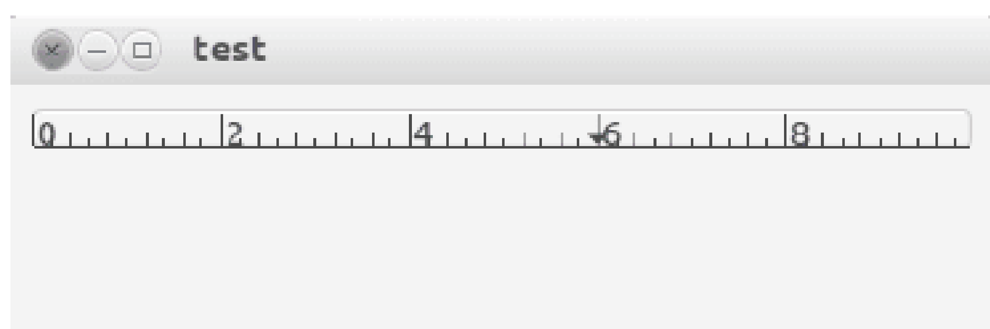


图 12.12　在窗口中添加横向标尺

实例的应用代码如下：

```
1       #include <gtk/gtk.h>
2
3       int main(int argc,char *argv[ ])
4       {
5           GtkWidget *window;
```

```
6        GtkWidget *box;
7        GtkWidget *hrule;
8        char title[]="test";
9
10       gtk_init(&argc,&argv);   //初始化函数
11       window = gtk_window_new(GTK_WINDOW_TOPLEVEL);      //创建窗体
12       gtk_window_set_title(GTK_WINDOW(window), title); //设置窗体的标题为 title 指向的字符串
13       gtk_container_set_border_width(GTK_CONTAINER(window),10);
14       gtk_widget_set_usize(GTK_WINDOW (window),400,100);        //设置窗体的大小
15       gtk_widget_set_uposition(GTK_WINDOW(window),200,200);     //设置窗体的起始坐标位置
16       gtk_signal_connect(GTK_OBJECT(window),"delete_event",G_CALLBACK(gtk_main_quit),NULL);
17       //登记窗体 delete_event 信号的回调函数
18
19       box = gtk_vbox_new(FALSE,0);                              //添加一个组合盒
20       hrule = gtk_hruler_new();                                 //创建标尺
21       gtk_ruler_set_metric(GTK_RULER(hrule),GTK_PIXELS);        //设置标尺的单位
22       gtk_ruler_set_range(GTK_RULER(hrule),0,10,0,10);   //设置标尺的跨度和指示器初始位置
23       gtk_box_pack_start(GTK_BOX(box),hrule,FALSE,FALSE,0);
24       gtk_container_add(GTK_CONTAINER(window),box);
25
26       gtk_widget_show(box);
27       gtk_widget_show(hrule);
28       gtk_widget_show(window);
29       gtk_main();
30   }
```

12.4 在 GTK+中使用组合构件

GTK+的组合构件通常是由多个基础构件结合而成，拥有更加强大的功能，常见的组合构件有对话框、组合框、日历构件等。

12.4.1 对话框

对话框构件是一个预先安装了几个构件的窗口构件，对其对应的结构体定义说明如下：

```
struct GtkDialog{
    GtkWindow window;
    GtkWidget *vbox;
    GtkWidget *action_area;
};
```

从该结构体中可以看到对话框构件首先创建一个窗口，然后在顶部放入了一个组合盒 GtkVbox，然后加入了一个活动区 action_area，这个活动区在本质上也是一个组合盒，这是一个横向的组合盒，可以在该组合盒中添加按键。

在 GTK+中使用函数 gtk_dialog_new 来创建对话框构件，函数没有参数值，其返回值是一个指向对话框的指针。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_dialog_new(void);
```

【例 12.14】在 GTK+中使用对话框控件实现退出提示

例 12.14 是一个使用对话框控件实现退出提示的实例，当用户单击如图 12.13 所示的窗口中的“exit”按钮时，将弹出对话框让用户选择“YES”和“NO”，应用代码在按钮的回调函数 button_deal 中加入了对对话框的操作。



图 12.13　退出确认对话框实例

实例的应用代码如下：

```
1    #include <gtk/gtk.h>
2
3    //按键的回调处理函数
4    void destroy(GtkWidget *widget,gpointer *data)
5    {
6        gtk_widget_destroy(GTK_WIDGET(data));
7    }
8
9    //按键的回调函数
10   gint button_deal(GtkWidget *widget,gpointer gdata)
11   {
12       GtkWidget *button;
13       GtkWidget *dialog;
14
15       dialog = gtk_dialog_new();      //新建对话框
16       button = gtk_button_new_with_label("YES");
17       gtk_box_pack_start(GTK_BOX(GTK_DIALOG(dialog)->action_area),button,TRUE,TRUE,0);
         //button 到对话框的操作区
18       gtk_signal_connect(GTK_OBJECT(button),"clicked",G_CALLBACK(gtk_main_quit),NULL);
19       gtk_widget_show(button);
20
21       button = gtk_button_new_with_label("NO");
22       gtk_box_pack_start(GTK_BOX(GTK_DIALOG(dialog)->action_area),button,TRUE,TRUE,0);
         //button 到对话框的操作区
23       gtk_signal_connect(GTK_OBJECT(button),"clicked",G_CALLBACK(destroy),dialog);
24       gtk_widget_show(button);
25       gtk_widget_show(dialog);
26   }
27
28
29   int main(int argc,char *argv[ ])
```

```
30      {
31          GtkWidget *window;
32          GtkWidget *button;          //窗口和按键的对应指针
33          char title[]="test";
34          gtk_init(&argc,&argv);   //初始化函数
35          window = gtk_window_new(GTK_WINDOW_TOPLEVEL); //创建窗体
36          gtk_window_set_title(GTK_WINDOW(window), title);
            //设置窗体的标题为 title 指向的字符串
37          gtk_widget_set_usize(GTK_WINDOW (window),300,150);    //设置窗体的大小
38          gtk_widget_set_uposition(GTK_WINDOW(window),200,200); //设置窗体的起始坐标位置
39          gtk_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
40          //登记窗体 delete_event 信号的回调函数
41
42          button = gtk_button_new_with_label("Exit");
43          gtk_container_add(GTK_CONTAINER(window),button);
44          gtk_signal_connect(GTK_OBJECT(button),"clicked",G_CALLBACK(button_deal),NULL);
45          gtk_widget_show(button);
46          gtk_widget_show(window);
47          gtk_main();
48      }
```

12.4.2　组合框

GTK 中的组合框和对话框类似，也是多个其他构件的集合，对于用户而言可以把其看作文本输入构件和下拉菜单的集合，对其对应的定义结构体说明如下：

```
struct _GtkCombo
{
  GtkHBox hbox;
  GtkWidget *entry;
  GtkWidget *button;
  GtkWidget *popup;
  GtkWidget *popwin;
  GtkWidget *list;
};
```

GTK+中通常使用 gtk_combo_new 函数来创建一个组合框，函数没有参数，其返回值是指向新建组合框的指针。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_combo_new(void);
```

组合框提供了一个文本输入构件，用户可以使用函数 gtk_entry_set_text 对这个文本输入框的内容直接进行操作，例如：

```
gtk_entry_set_text(GTK_ENTRY(GTK_COMBO(combo)->entry),"plz check");
```

如果要向组合框的下拉菜单中加入内容，则应该首先建立一个链表，然后使用 gtk_combo_set_popdown_strings 函数来完成对应的操作，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_combo_set_popdown_strings( GtkCombo *combo, GList *strings );
```

其中参数 combo 为指向组合框的指针，strings 为指向待加入的链表，函数没有返回值，对建立链表的代码说明如下：

```
GList *glist=NULL;
glist = g_list_append(glist, "String 1");
glist = g_list_append(glist, "String 2");
glist = g_list_append(glist, "String 3");
glist = g_list_append(glist, "String 4");
```

【例 12.15】在 GTK+中使用组合框控件实现课程选择

例 12.15 是一个组合框的应用实例，利用下拉菜单实现数学、语文、外语课程的选择，并且还添加了“请选择”的提示，用户可以通过单击选择其中一门课程，其运行如图 12.14 所示，应用代码使用了一个链表 glist 用于存放课程的类别。

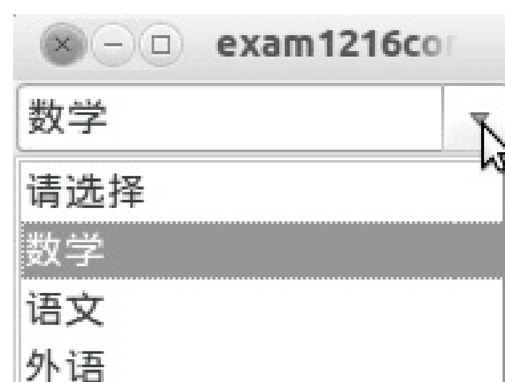


图 12.14　组合框的应用实例

实例的应用代码如下：

```
1    #include <gtk/gtk.h>
2
3    int main(int argc,char *argv[])
4    {
5        GtkWidget *window;
6        GtkWidget *combo;
7        GList *glist = NULL;
8        gtk_init(&argc,&argv);
9        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
10       gtk_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
11       combo = gtk_combo_new( );           //创建组合框
12       glist = g_list_append(glist,"请选择"); //添加字符串到链表
13       glist = g_list_append(glist,"数学");
14       glist = g_list_append(glist,"语文");
15       glist = g_list_append(glist,"外语");
16       gtk_combo_set_popdown_strings(GTK_COMBO(combo),glist);
17       gtk_container_add(GTK_CONTAINER(window),combo);
18       gtk_widget_show(combo);
19       gtk_widget_show(window);
20       gtk_main();
21   }
```

12.4.3 微调构件

微调构件由一个文本输入框和旁边的向上、向下两个按钮组成，单击向上按钮会让文本输入框中的数值变大，单击向下按钮会让文本输入框中的数值变小。微调按钮中的数值可以有小数点，并且这个数值可以按照配置变大或者变小，并且如果长时间单击同一个按钮不释放会加速这个数值的变化。

和微调构件相关的是一个微调对象，该对象用于维护微调构件中按钮的取值范围，可以使用函数 gtk_adjustment_new 来创建这个微调对象，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkObject *gtk_adjustment_new(gflost value, gfloat lower, gflost upper, gfloat step_increment, gfloat page_increment, gfloat page_size);
```

对函数中的各个参数说明如下。

- value：微调按钮的初始化值（显示值）。
- lower：构件允许的最小值。
- upper：构件允许的最大值。
- step_increment：当鼠标左键单击时构件一次增加/减少的值。
- page_increment：当鼠标右键单击时构件一次增加/减少的值。
- page_size：没有意义的参数值。

当创建完微调对象之后即可调用 gtk_spin_button_new 函数来创建微调按钮构件，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_spin_button_new(GtkAdjustment *adjustment, gfloat clim_rate, guint digits);
```

其中参数 adjustment 用于指向微调对象，clim_rate 为微调构件变化的加速度，这是一个介于 0 和 1 之间的值，digits 用于设置显示数值的小数位。

GTK+还提供了一些其他函数，用于对微调按钮进行操作，对这些函数说明如下。

- gtk_spin_button_set_adjustment 函数：用于设置微调按钮的值，其中参数 spin_button 为指向微调按钮的指针，adjustment 为指向微调对象的指针，函数没有返回值，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_spin_button_set_adjustment( GtkSpinButton *spin_button, GtkAdjustment *adjustment );
#include <gtk/gtk.h>
GtkAdjustment adjustment(GtkSpinButton *spin_button );
```

- gtk_spin_button_set_digits 函数：用于修改显示数值的小数位，参数 spin_button 为指向微调按钮的指针，参数 digits 为显示数值的小数位，函数没有返回值。对标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_spin_button_set_digits( GtkSpinButton *spin_button,guint digits) ;
```

- gtk_spin_button_set_value 函数：用于修改当前微调按钮的显示值，其中参数 spin_button 为指向微调按钮的指针，而参数 value 为修改值，函数没有返回值。对其标准调用格式说明如下。

```
#include <gtk/gtk.h>
void gtk_spin_button_set_value( GtkSpinButton *spin_button, gfloat value );
```

- gtk_spin_button_get_value_as_float 和 gtk_spin_button_get_value_as_int 函数：用于获得微调按钮的当前值，这两个函数的参数都是指向微调按钮的指针，gtk_spin_button_get_value_as_float 函数的返回值是一个 float 类型的变量，gtk_spin_button_get_value_as_int 函数的返回值是一个 int 类型的变量。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
gfloat gtk_spin_button_get_value_as_float( GtkSpinButton *spin_button );
gint gtk_spin_button_get_value_as_int( GtkSpinButton *spin_button );
```

- gtk_spin_button_set_wrap 函数：用于设置是否让微调按钮在 upper 和 lower 之间循环，也就是如果达到了最大值继续变化则变为最小值，或者达到最小值则变为最大值，参数 spin_button 为指向微调按钮的指针，wrap 有 TRUE 和 FALSE 两种选值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_spin_button_set_wrap( GtkSpinButton *spin_button,gboolean wrap );
```

【例 12.16】在 GTK+中使用微调控件实现年份信息的选择

例 12.16 是一个微调控件的应用实例，其建立了一个如图 12.15 所示的窗口，可以用于选择当前的年份信息，使用微调控件的箭头可以实现对年份的翻页，通过单击“确定”按钮可以确认当前的年份选择，应用代码在按钮的回调函数 button_deal 中对按钮的单击事件调用 gtk_spin_button_get_value_as_int 函数来获得微调按键的当前值，以确定用户选择的年份。

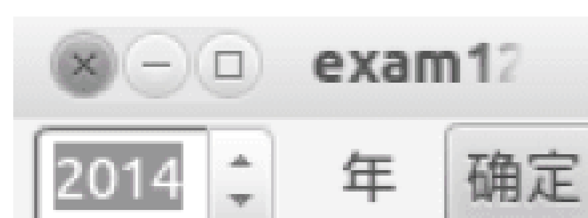


图 12.15　一个微调构件的应用实例

实例的应用代码如下：

```
1   #include <gtk/gtk.h>
2
3   GtkWidget *spin;
4
5   //处理按键事件的回调函数
6   void button_deal(GtkWidget *widget,gpointer *data)
```

```
7    {
8          gint year;
9          year = gtk_spin_button_get_value_as_int(GTK_SPIN_BUTTON(spin));
         //获得微调按钮的当前值
10         g_print("Year:%d",year);     //在终端输出当前值
11   }
12
13   int main(int argc,char *argv[])
14   {
15         GtkWidget *window;
16         GtkWidget *box;
17         GtkWidget *label;
18         GtkWidget *button;
19         GtkObject *adjustment;
20         gtk_init(&argc,&argv);
21         window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
22         gtk_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
23         box = gtk_hbox_new(FALSE,10);                            //创建组合盒
24         gtk_container_add(GTK_CONTAINER(window),box);            //将组合盒加入窗体
25         adjustment = gtk_adjustment_new(2014,1900,2100,1,1,0);   //创建微调对象
26         spin = gtk_spin_button_new(GTK_ADJUSTMENT(adjustment),0.5,0);    //创建微调按钮
27         gtk_box_pack_start(GTK_BOX(box),spin,TRUE,TRUE,5);       //将微调按钮加入组合盒
28         gtk_widget_show(spin);
29         label = gtk_label_new("年");
30         gtk_box_pack_start(GTK_BOX(box),label,TRUE,TRUE,0);
31         gtk_widget_show(label);
32         button = gtk_button_new_with_label("确定");
33         gtk_box_pack_start(GTK_BOX(box),button,TRUE,TRUE,0);
34         gtk_signal_connect(GTK_OBJECT(button),"clicked",GTK_SIGNAL_FUNC(button_deal),NULL);
35         gtk_widget_show(button);
36         gtk_widget_show(box);
37         gtk_widget_show(window);
38         gtk_main();
39   }
```

12.4.4 日历构件

GTK+提供了一个日历构件，以供对日期相关的应用代码使用，在其中可以很方便地选择指定的包括年、月、日的日期信息。

GTK+提供了如下函数用于对日历构件进行操作。

- gtk_calendar_new 函数：用于创建一个新的日历控件，其没有参数和返回值，对标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_calendar_new();
```

- gtk_calendar_freeze 和 gtk_calendar_thaw 函数：用于冻结和解冻日历构件，这是为了避免在修改日历构件的过程中不停地更新导致构件显示效果的闪烁，所以在修改过程中可

以先将构件冻结，然后再解冻，这样就可以实现整个过程中只更新一次的显示效果，函数的参数均为指向待冻结和解冻的日历构件指针，函数没有返回值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_calendar_freeze( GtkCalendar *Calendar );
void gtk_calendar_thaw ( GtkCalendar *Calendar );
```

- gtk_calendar_display_options 函数：用于设置日历构件的外观和操作方式，其中参数 calendar 为指向日历构件的指针，flags 参数用于设置日历构件的显示风格，有如下 5 种取值：GTK_CALENDAR_SHOW_HEADING 用于设置在绘制日历构件时是否显示月和年信息；GTK_CALENDAR_SHOW_DAY_NAMES 用于设置是否使用星期的字母缩写，如 MON、TUE 等；GTK_CALENDAR_NO_MONTH_CHANCE 用于设置当前构件显示的月份信息是否能被改变；GTK_CALENDAR_SHOW_WEEK_NUMBERS 用于设置是否在日历构件的左侧显示全年的周序号，整个年可以划分为 52 个周，其中 1 月 1 日是第一周的第一天；GTK_CALENDAR_WEEK_START_MONDAY 用于设置每周的第一天是周一还是周日。函数没有返回值，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_calendar_display_options( GtkCalendar *calendar, GtkCalendarDisplayOptions flags );
```

- gtk_calendar_select_month 和 gtk_calendar_select_day 函数：用于设置当前日历构件的日期信息，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
gint gtk_calendar_select_month( GtkCalendar *calendar,guint month, guint year );
void gtk_calendar_select_day( GtkCalendar *calendar, guint day );
```

- gtk_calendar_select_month 函数，用于设置日历构件的年份和月份信息，其中参数 calendar 为指向设置日历的指针，参数 month 和 year 分别为期待设置的年份和月份数据，函数的返回值表明是否设置成功，如果设置失败则会返回 FALSE，否则返回 TRUE。
- gtk_calendar_select_day 函数用于设置日历构件的日信息，其中参数 calendar 为指向设置日历的指针，参数 day 为待设置的日信息，函数没有返回值，如果设置成功则设置的该日期被选中。
- gtk_calendar_mark_day、gtk_calendar_unmark_day 和 gtk_calendar_clear_marks 函数：对一个月中的指定日期进行标记，同一个月中可以有多个日期被标记，并且在日历构件中高亮显示。函数的参数 calendar 均为指向待标记日期的日历构件指针，day 为待标记的日期，函数 gtk_calendar_mark_day 用于标记日期，如果成功则返回 TRUE，否则返回 FALSE；函数 gtk_calendar_unmark_day 用于取消标记日期，如果成功则返回 TRUE，否则返回 FALSE; 函数 gtk_calendar_clear_marks 用于取消当前日历构件中的所有标记日期。对此三个函数的标准调用格式说明如下：

```
#include <gtk/gtk.h>
```

```
gint gtk_calendar_mark_day( GtkCalendar *calendar,guint day);
gint gtk_calendar_unmark_day( GtkCalendar *calendar,guint day);
void gtk_calendar_clear_marks( GtkCalendar *calendar);
```

- gtk_calendar_get_date 函数：用于获取当前用户选择的日期信息，函数的参数 calendar 为指向日历构件的指针，参数 year、month 和 day 用于存放返回的日期信息，函数没有返回值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_calendar_get_date( GtkCalendar *calendar,guint *year,guint *month,guint *day );
```

日历构件还提供了一些信号以供用户使用，这些信号包括：

- month_changed：被选择的月份发生变化。
- day_selected：选择日期发生变化。
- day_selected_double_click：选中一个日期并且被双击。
- prev_month：选择上一个月。
- next_month：选择下一个月。
- prev_year：选择上一年。
- next_year：选择下一年。

【例 12.17】在 GTK+中使用日历控件实现日期选择

例 12.17 是一个日历控件的应用实例，其建立了一个如图 12.16 所示的日历选择对话框，可以更改选择当前的日历信息，当选择一个日期并且单击确定之后会在终端中输出对应的信息。应用代码在确定按钮的回调函数中调用了 g_print 函数，在终端中输出当前选择的日期信息。



图 12.16 日历构件的应用

实例的应用代码如下：

```
1	#include <gtk/gtk.h>
2
3	GtkWidget *calendar;		//日历构件
4
5	//按键处理的回调函数
6	void button_event(GtkWidget *widget,gpointer *data)
7	{
```

```
8       guint year;
9       guint month;
10      guint day;
11      gtk_calendar_get_date(GTK_CALENDAR(calendar),&year,&month,&day);
       //取得选择的年月日
12      g_print("Year:%d Month:%d Day:%d\n",year,month,day);     //在终端输出
13    }
14
15    int main(int argc,char *argv[ ])
16    {
17      GtkWidget *window;
18      GtkWidget *box;
19      GtkWidget *button;
20      gtk_init(&argc,&argv);
21      window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
22      gtk_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
23      box=gtk_vbox_new(FALSE,10);//建立组合盒
24      gtk_container_add(GTK_CONTAINER(window),box);              //将组合盒加入窗体
25      calendar = gtk_calendar_new();                             //建立日历构件
26      gtk_box_pack_start(GTK_BOX(box),calendar,TRUE,TRUE,5);     //将日历构件加入组合盒
27      gtk_widget_show(calendar);                                 /*显示日历构件*/
28      button = gtk_button_new_with_label("确定");
29      gtk_box_pack_start(GTK_BOX(box),button,TRUE,TRUE,0);
30      gtk_signal_connect(GTK_OBJECT(button),"clicked",GTK_SIGNAL_FUNC(button_event),NULL);
31      gtk_widget_show(button);
32      gtk_widget_show(box);
33      gtk_widget_show(window);
34      gtk_main();
35    }
```

单击切换并且选中一个新的日期，可以在终端中看到对应的信息输出：

```
Year:2014 Month:4 Day:14
Year:2014 Month:4 Day:30
```

12.4.5 文件选择构件

在 GTK+图形编程中可能要打开对话框进行文件的选择，此时可以使用文件选择构件，其带有一些基础控制按钮，能极大地提高编程效率。文本选择构件自身能提供一个窗体，所以不应该、也不能将其放入另外一个窗体中，通常来说应用代码会在一个“打开文件”按钮或者菜单项中打开这个文件，以便选择构件，对其内部定义结构如下：

```
typedef struct {
GtkWidget *dir_list;
GtkWidget *file_list;
GtkWidget *selection_entry;
GtkWidget *selection_text;
GtkWidget *main_vbox;
GtkWidget *ok_button;
```

```
GtkWidget *cancel_button;
GtkWidget *help_button;
GtkWidget *history_pulldown;
GtkWidget *history_menu;
GList     *history_list;
GtkWidget *fileop_dialog;
GtkWidget *fileop_entry;
gchar     *fileop_file;
gpointer   cmpl_state;
GtkWidget *fileop_c_dir;
GtkWidget *fileop_del_file;
GtkWidget *fileop_ren_file;
GtkWidget *button_area;
GtkWidget *action_area;
}GtkFileSelection;
```

GTK+提供了一系列函数用于对文件选择构件进行操作，对这些函数说明如下。

- gtk_file_selection_new 函数：用于创建一个文件选择构件，函数的参数为文件选择构件的标题，函数的返回值为一个指向文件选择构件的指针。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_file_selection_new( gchar *title );
```

- gtk_file_selection_set_filename 函数：用于设置一个默认的指向文件名称，函数的参数 filesel 为一个指向文本选择框的指针，filename 为默认的指向文件名称，函数没有返回值，通常来说该函数用于设置打开文件选择框时的默认文件值或者文件夹。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_file_selection_set_filename( GtkFileSelection *filesel, gchar *filename );
```

- gtk_file_selection_get_filename 函数：用于获取当前文本选择框中用户选中的文件名称，其中参数 filesel 为指向文件选择框的指针，函数返回值为用户选中的文件名称字符串。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
gchar *gtk_file_selection_get_filename( GtkFileSelection *filesel );
```

【例 12.18】在 GTK+中使用文件选择控件实现文件管理对话框

例 12.18 是使用文件选择构件在如图 12.17 所示的 Linux 图形界面中选中一个文件，并且在终端中输出选中文件的完整路径实例，其通常会用于实现文件管理对话框，如“打开”、“保存”、“另存为”等。应用代码构造了一个 OpenFile 函数，用于对“打开文件”事件进行处理，使用 g_printf 函数在终端中输出了当前选中的文件信息，而该函数在按钮的回调函数中被调用。



图 12.17 文件选择构件的应用

实例的应用代码如下：

```
1     #include <gtk/gtk.h>
2
3     GtkWidget *FileSelection;      //文件选择构件
4
5     //在终端中输出当前选择的文件名称
6     void OpenFile(GtkWidget *widget,gpointer *data)
7     {
8        g_print("%s\n",gtk_file_selection_get_filename(GTK_FILE_SELECTION(FileSelection)));
9     }
10
11    //按键处理的回调函数
12    void button_event(GtkWidget *widget,gpointer *data)
13    {
14       FileSelection=gtk_file_selection_new("选择文件");      //创建文件选择构件
15       gtk_file_selection_set_filename(GTK_FILE_SELECTION(FileSelection),"*.txt");
16       gtk_signal_connect(GTK_OBJECT(GTK_FILE_SELECTION(FileSelection)->ok_button),"clicked",
         GTK_SIGNAL_FUNC(OpenFile),NULL);
17       //捕捉打开按钮的“clicked”信号
18       gtk_widget_show(FileSelection);
19    }
20
21
22    int main(int argc,char *argv[ ])
23    {
24       GtkWidget *window;
25       GtkWidget *button;
26       gtk_init(&argc,&argv);
27       window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
28       gtk_widget_set_size_request(window,200,100);/*调整窗口大小*/
29       gtk_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
30       button=gtk_button_new_with_label("打开文件");/*常见按钮*/
31       gtk_signal_connect(GTK_OBJECT(button),"clicked",GTK_SIGNAL_FUNC(button_event),NULL);
```

```
32          gtk_container_add(GTK_CONTAINER(window),button);
33          gtk_widget_show(button);
34          gtk_widget_show(window);
35          gtk_main();
36      }
```

当选中一个文件的时候，会在终端中看到其完整的路径输出：

```
/home/alloy/linuxc/chapter12/exam1203owncallback.c
/home/alloy/linuxc/chapter12/exam1201window.c
/home/alloy/linuxc/chapter12/exam1204button
```

12.4.6　按钮盒

当需要在 GTK 的应用代码中使用多个按钮时，可以使用按钮盒构件，其可以很方便地快速布置一组按钮，按钮盒有横向和纵向两种不同的格式。

GTK+同样提供了一系列函数用于对按钮盒进行相应的操作。

gtk_hbutton_box_new 和 gtk_vbutton_box_new 函数用于创建一个按钮盒，对其标准调用格式说明如下：

```
#include    <gtk/gtk.h>
GtkWidget *gtk_hbutton_box_new( void );
GtkWidget *gtk_vbutton_box_new( void );
```

其中 gtk_hbutton_box_new 函数用于创建一个横向的按钮盒，gtk_vbutton_box_new 用于创建一个纵向的按钮盒，然后就可以使用 gtk_container_add 将按钮加入按钮盒。按钮在按钮盒中是间隔排放的，可以使用函数 gtk_hbutton_box_set_spacing_default 和函数 gtk_vbutton_box_set_spacing_default 修改这些按钮的间距，也可以使用 gtk_hbutton_box_get_spacing_default 和 gtk_vbutton_box_get_spacing_default 函数来获得按钮的间距，对这些函数的标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_hbutton_box_set_spacing_default( gint spacing );
void gtk_vbutton_box_set_spacing_default( gint spacing );
gint gtk_hbutton_box_get_spacing_default( void );
gint gtk_vbutton_box_get_spacing_default( void );
```


注　意

以上函数的设置参数 spacing 以及返回值的单位都是像素。

函数 gtk_hbutton_box_set_layout_default 和 gtk_vbutton_box_set_layout_default 用于设置按钮盒中按钮的布局；函数 gtk_hbutton_box_get_layout_default 和 gtk_vbutton_box_get_layout_default 用于取得按钮盒中按钮的布局，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_hbutton_box_set_layout_default( GtkButtonBoxStyle layout );
void gtk_vbutton_box_set_layout_default( GtkButtonBoxStyle layout );
GtkButtonBoxStyle gtk_hbutton_box_get_layout_default( void );
```

```
GtkButtonBoxStyle gtk_vbutton_box_get_layout_default( void );
```

函数的参数 layout 用于设置按钮盒的布局风格，有如下的取值可以选择：GTK_BUTTONBOX_DEFAULT_STYLE、GTK_BUTTONBOX_SPREAD、GTK_BUTTONBOX_EDGE、GTK_BUTTONBOX_START 和 GTK_BUTTONBOX_END。

【例 12.19】在 GTK+中使用按钮盒控件实现简单应用程序界面

例 12.19 是使用按钮盒控件实现一个简单应用程序界面的应用，如图 12.18 所示的界面中包括了三个按钮，分别是“打开”、“关闭”和“帮助”。应用代码把这三个按钮放到了一个横向的按钮盒中，需要注意的是并没有添加这三个按钮的回调函数，用户可以自行修改这三个按钮的标签和添加合适的回调函数以实现其他的功能。



图 12.18　有三个按钮的横向按钮盒

实例的应用代码如下：

```
1     #include <gtk/gtk.h>
2
3     int main(int argc,char *argv[ ])
4     {
5        GtkWidget *window;
6        GtkWidget *button_box;
7        GtkWidget *button;
8        gtk_init(&argc,&argv);
9        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
10       gtk_widget_set_size_request(window,300,50);
11       gtk_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
12       button_box = gtk_hbutton_box_new();       //创建按钮盒构件
13       gtk_hbutton_box_set_spacing_default(5); //设置按钮间距
14       gtk_hbutton_box_set_layout_default(GTK_BUTTONBOX_SPREAD);   //设置按钮盒布局
15       gtk_container_add(GTK_CONTAINER(window),button_box);        //将按钮盒构件加入窗体
16       gtk_widget_show(button_box);
17       button=gtk_button_new_with_label("打开");
18       gtk_container_add(GTK_CONTAINER(button_box),button);        //将按钮加入按钮盒构件
19       gtk_widget_show(button);
20       button=gtk_button_new_with_label("关闭");
21       gtk_container_add(GTK_CONTAINER(button_box),button);
22       gtk_widget_show(button);
23       button=gtk_button_new_with_label("帮助");
24       gtk_container_add(GTK_CONTAINER(button_box),button);
25       gtk_widget_show(button);
26       gtk_widget_show(window);
27       gtk_main();
28    }
```

12.4.7　框架

框架是一个自带风格和位置可变的标签，并且可以用于在盒子中封装一个或者多个构件的构件，在实际应用中通常用于将多个构件进行分类组合。

GTK+提供了如下的函数用于对框架进行操作。

- gtk_frame_set_label 函数：用于创建一个框架，函数的参数 label 是一个用于显示的标签字符串，该标签默认放在框架的左上角，如果不想显示该标签，则应该将 NULL 传递给 label。函数的返回值是一个指向新建框架的指针。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_frame_new( const gchar *label );
```

- gtk_frame_set_label 函数：用于修改框架的标签，函数的参数 frame 为指向待修改标签的框架指针，参数 label 为待设置的标签字符串，函数没有返回值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_frame_set_label(GtkFrame *frame,const gchar *label);
```

- gtk_frame_set_label_align 函数：用于设置框架的标签位置，函数的参数 frame 为指向待设置标签位置的框架指针，参数 xalign 用于设置标签字符串在框架上部水平线的位置，其是一个 0.0~1.0 之间的取值，当该值为 0.0 时该标签被放在框架构件的左上角。函数没有返回值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_frame_set_label_align(GtkFrame *frame,gfloat xalign,gfloat yalign);
```

- gtk_frame_set_label_align 函数：用于修改框架构件的轮廓风格，其中参数 frame 为指向待修改风格框架的指针，type 为框架的风格取值，有 GTK_SHADOW_NONE、GTK_SHADOW_IN、GTK_SHADOW_OUT、GTK_SHADOW_ETCHED_IN（缺省值）和 GTK_SHADOW_ETCHED_OUT 这几个取值，函数没有返回值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_frame_set_shadow_type(GtkFrame *frame,GtkShadowType type);
```

【例 12.20】在 GTK+中使用框架控件实现性别选择

例 12.20 是一个使用框架将一个标签和一个单选框组合到一起放入框架的应用，用户可以在其中进行性别选择，如图 12.19 所示。

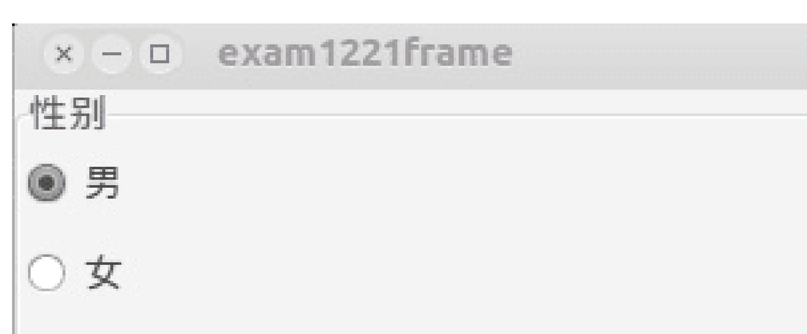


图 12.19　框架的应用实例

实例的应用代码如下：

```
1     #include <gtk/gtk.h>
2
3     int main(int argc,char *argv[ ])
4     {
5        GtkWidget *window;
6        GtkWidget *frame;
7        GtkWidget *button;
8        GtkWidget *box;
9        GSList *group = NULL;    //定义组
10       gtk_init(&argc,&argv);
11       window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
12       gtk_widget_set_size_request(window,300,100);
13       gtk_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
14       frame = gtk_frame_new("性别");      //创建框架构件
15       gtk_container_add(GTK_CONTAINER(window),frame);    //将框架构件加入窗体
16       gtk_widget_show(frame);
17       box = gtk_vbox_new(FALSE,0);    //创建组合框
18       gtk_container_add(GTK_CONTAINER(frame),box);    //将组合框加入框架构件
19       gtk_widget_show(box);
20       button = gtk_radio_button_new_with_label(group,"男");    //创建按钮
21       group = gtk_radio_button_group(GTK_RADIO_BUTTON(button)); //将按钮加入组合框
22       gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,5);
23       gtk_widget_show(button);
24       button = gtk_radio_button_new_with_label(group,"女");
25       group = gtk_radio_button_group(GTK_RADIO_BUTTON(button));
26       gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,5);
27       gtk_widget_show(button);
28       gtk_widget_show(window);
29       gtk_main();
30    }
```

12.4.8　文本框

文本框是 GTK+中最常用的输入构件，其是一个复合构件，可以分为如下几个部分。

- GtkTextView：代表了窗口中可见的文本框，用来显示 GtkTextBuffer。
- GtkTextBuffer：文本框中正文的缓冲区，文本框文字的插入、删除都是对这一类变量进行操作。
- GtkTextIter：保存文字在缓冲区中的位置结构。
- GtkTextMark：缓冲区中的修改记录。
- GtkTextTag：用来给指定的文字添加一些标记，改变指定区域的文字显示效果，例如字体的颜色、大小的改变。
- GtkTextTagTable：是 GtkTextTag 标记的集合表。

GTK+提供了函数 gtk_text_view_new 和 gtk_text_view_new_with_buffer 来创建文本框，对其标

准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_text_view_new(void);
GtkWidget *gtk_text_view_new_with_buffer(GtkTextBuffer *buffer);
```

这两个函数的返回值都是指向新建文本框的指针，其中 buffer 为文本框对应的缓冲区指针，这个缓冲区可以被多个构件共享，其取值可以是 NULL，如果没有指定这个缓冲区，则会分配给文本框一个默认的缓冲区，可以使用函数 gtk_text_view_set_buffer 来设置这个缓冲区，也可以使用函数 gtk_text_view_get_buffer 来获取这个缓冲区，其中参数 text_view 是指向待获取缓冲区文本框的指针，buffer 是指向缓冲区的指针。对这两个函数的标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_text_view_set_buffer(GtkTextView *text_view,GtkTextBuffer *buffer);
GtkTextBuffer* gtk_text_view_get_buffer (GtkTextView *text_view);
```

每个显示出来的文本框（GtkTextView）都应该对应一个缓冲区（GtkTextBuffer），可以使用函数 gtk_text_buffer_new 来创建文本框缓冲区，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkTextBuffer* gtk_text_buffer_new(GtkTextTagTable *table);
```

函数的返回值是一个指向文本框缓冲区的指针，其中参数 table 是文本框的标签盒，如果该参数为空，则 GTK+会创建一个默认的标签盒，用户可以使用 gtk_text_buffer_get_tag_table 函数来获得这个标签盒，其中参数 buffer 为指向文本框缓冲区的指针，函数的返回值是文本框缓冲区的标签盒。对该函数的标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkTextTagTable* gtk_text_buffer_get_tag_table(GtkTextBuffer *buffer);
```

对文本框中的字符串操作是通过向缓冲区中的数据操作完成的，GTK+提供了如下函数用于对这个缓冲区进行操作。

- gtk_text_buffer_get_bounds 函数：获得当前缓冲区中开始和结束位置的 ITER，其中参数 buffer 是指向文本框缓冲区的指针，start 和 end 则分别放置了缓冲区起始和结束位置的 iter，函数没有返回值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_text_buffer_get_bounds(GtkTextBuffer *buffer,GtkTextIter *start, GtkTextIter *end);
```

- gtk_text_buffer_insert 函数：向文本框的缓冲区中插入字符串，其中 buffer 为指向文本缓冲区的指针，iter 为字符串的插入位置，text 为指向待插入字符串的指针，len 为待插入字符串的长度，如果该值为“-1”，则表示插入 text 所指向字符串的所有内容，函数没有返回值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_text_buffer_insert(GtkTextBuffer *buffer,GtkTextIter *iter,const gchar *text,gint len);
```

- gtk_text_buffer_delete 函数：用于删除缓冲区的数据，其中 buffer 为指向文本缓冲区的指针，start 和 end 分别为指向文本框文字的开始位置和结束位置的 iter。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_text_buffer_delete(GtkTextBuffer *buffer,GtkTextIter *start,GtkTextIter *end);
```

- gtk_text_buffer_set_text 函数：将原缓冲区的内容删除，然后填充指定的内容，其中 buffer 为指向缓冲区的指针，text 为指向待设置字符串的缓冲区，len 为字符串的长度，函数没有返回值。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
void gtk_text_buffer_set_text(GtkTextBuffer *buffer,const gchar *text,gint len);
```

用户可以利用函数 gtk_text_buffer_get_text 来获得文本框缓冲区的内容，对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
gchar* gtk_text_buffer_get_text(GtkTextBuffer *buffer,const GtkTextIter *start,const GtkTextIter *end,
 gboolean include_hidden_chars);
```

其中参数 buffer 为指向文本缓冲区的指针，start 和 end 分别是文本框文字开始和结束的 iter，include_hidden_chars 用于设置是否包括隐藏字符，函数返回值是指向文本缓冲区内容的指针。

【例 12.21】在 GTK+中使用文本框控件实现文本输出

例 12.21 是一个文本框的应用实例，其可以在如图 12.20 所示的文本框中输入一个字符串，当单击“确定”按钮会在终端中打印出当前输入的字符串；应用代码在“确定”按钮的回调函数中使用 gtk_text_buffer_get_bounds 函数来获得当前输入字符串缓冲区的位置，然后调用 gtk_text_buffer_get_text 函数将其放到字符串指针 text 中，最后调用 g_print 函数在终端中输出。



图 12.20　文本框的应用实例

实例的应用代码如下：

```
1       #include <gtk/gtk.h>
2
```

```
3    GtkWidget *text_view;
4    GtkTextBuffer *buffer;
5    GtkTextIter *Iter;
6
7    void button_event(GtkWidget *widget,gpointer *data)
8    {
9       gchar *text;
10      GtkTextIter start,end;
11      gtk_text_buffer_get_bounds(GTK_TEXT_BUFFER(buffer),&start,&end);
12      //获得缓冲区开始和结束位置的 Iter
13      const GtkTextIter s=start,e=end;
14      text = gtk_text_buffer_get_text(GTK_TEXT_BUFFER(buffer),&s,&e,FALSE);
        /*获得文本框缓冲区文本*/
15      g_print("%s\n",text);
16   }
17
18   int main(int argc,char *argv[ ])
19   {
20      GtkWidget *window;
21      GtkWidget *button;
22      GtkWidget *box;
23      gtk_init(&argc,&argv);
24      window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
25      gtk_widget_set_size_request(window,300,200);
26      g_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
27      box = gtk_vbox_new(FALSE,0);
28      gtk_widget_show(box);
29      text_view = gtk_text_view_new();                //创建文本框构件
30      gtk_widget_set_size_request(text_view,300,170);
31      gtk_container_add(GTK_CONTAINER(window),box);
32      gtk_box_pack_start(GTK_BOX(box),text_view,FALSE,FALSE,0);
33      buffer=gtk_text_view_get_buffer(GTK_TEXT_VIEW(text_view));
34      gtk_widget_show(text_view);
35      button = gtk_button_new_with_label("确定");
36      gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,5);
37      gtk_signal_connect(GTK_OBJECT(button),"clicked",GTK_SIGNAL_FUNC(button_event),NULL);
38      gtk_widget_show(button);
39      gtk_widget_show(window);
40      gtk_main();
41   }
```

12.5 设计 GTK+的菜单

菜单是 GTK+应用程序中最常用的设计，通常来说一个完整应用程序必须包括一个位于图形界面顶部的菜单。其由菜单条（GtkMenuBar）和下拉菜单（GtkMenu）组成，需要注意的是在建立菜单条之间必须建立一个纵向组合框。

12.5.1 建立菜单

一个在 GTK+中建立菜单的完整流程如下：

01 使用函数 gtk_menu_bar_new 建立一个菜单条，并且将其添加到纵向组合框中。
02 使用函数 gtk_menu_item_new_with_label 建立带标号的菜单项。
03 使用函数 gtk_menu_bar_append 将建立好的菜单项添加到菜单条中。
04 如果菜单需要有子菜单，则分别建立包括子菜单在内的新菜单项，然后使用函数 gtk_menu_append 或者 gtk_menu_shell_append 函数将子菜单项加入到子菜单。
05 使用 gtk_menu_item_set_submenu 函数将子菜单项和对应的菜单项联系到一起。

对以上涉及的各个函数的标准调用格式说明如下：

```
#include <gtk/gtk.h>
GtkWidget *gtk_menu_bar_new(void);
```

gtk_menu_bar_new 函数用于建立一个新的菜单项目，函数没有参数，返回值是一个指向新菜单项的指针，建立一个标准的菜单并且将其放入纵向组合框的代码说明如下：

```
et_show(menubar);
```

gtk_menu_item_new_with_label 函数用于向菜单条中添加带标签的菜单项，其中参数 label 为菜单的标签，函数的返回值为指向菜单项的指针。

```
#include <gtk/gtk.h>
GtkWidget* gtk_menu_item_new_with_label(const gchar *label);
```

gtk_menu_shell_append 函数用于将菜单项加入菜单条，其中参数 child 为指向待加入菜单项的指针，menu 为指向目标菜单条的指针，函数没有返回值。

```
#include <gtk/gtk.h>
void gtk_menu_shell_append(GtkMenuShell *menu,GtkWidget *child);
```

【例 12.22】一个带子菜单的菜单栏应用

例 12.22 是一个建立如图 12.21 所示菜单的应用实例，其包括了 File、Edit、View、Insert 和 Tools 共 5 个菜单栏，其中 File 菜单栏还有 4 个子菜单，分别是“New”、“Open”、“Save”和“Exit”。应用代码在子函数 CreateMenu 中创建 File 菜单栏的子菜单，然后使用 gtk_menu_item_set_submenu 函数将子菜单和菜单栏连接起来。

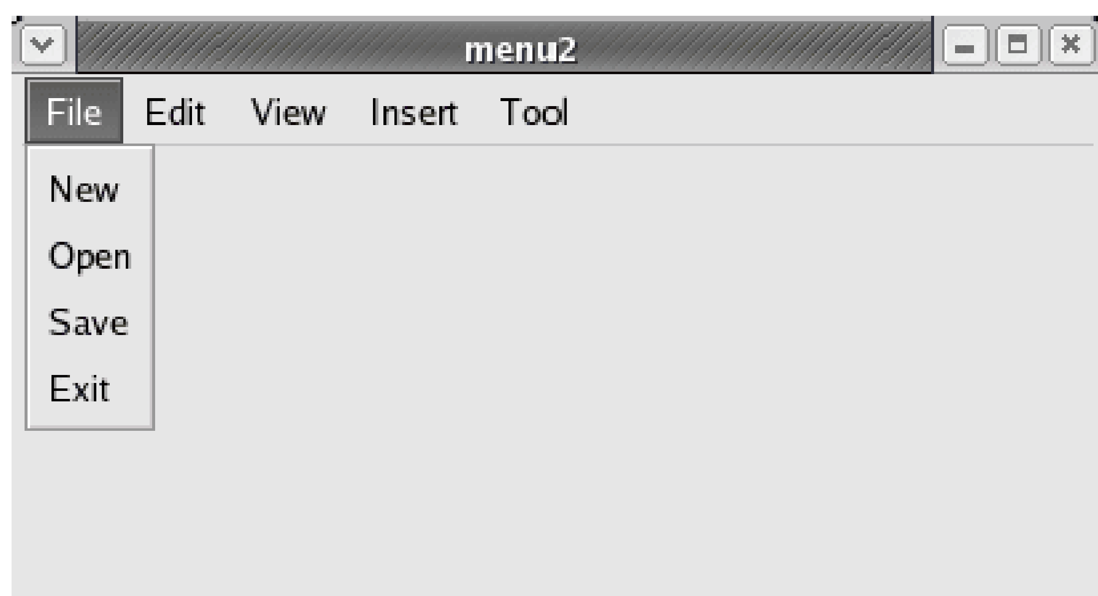


图 12.21 一个简单的菜单应用实例

实例的应用代码如下：

```
1    #include <gtk/gtk.h>
2
3    GtkWidget *CreateMenuItem(GtkWidget *MenuBar,char *test)
4    {
5       GtkWidget *MenuItem;
6       MenuItem = gtk_menu_item_new_with_label(test);   //创建菜单项
7       gtk_menu_shell_append(GTK_MENU_SHELL(MenuBar),MenuItem); //把菜单项加入菜单条
8       gtk_widget_show(MenuItem);
9       return MenuItem;
10   }
11
12   GtkWidget *CreateMenu(GtkWidget *MenuItem)
13   {
14      GtkWidget *Menu;                              //定义子菜单
15      Menu = gtk_menu_new();                        //创建子菜单
16      CreateMenuItem(Menu,"New");                   //调用创建菜单项函数
17      CreateMenuItem(Menu,"Open");
18      CreateMenuItem(Menu,"Save");
19      CreateMenuItem(Menu,"Exit");
20      gtk_menu_item_set_submenu(GTK_MENU_ITEM(MenuItem),Menu);
        //把父菜单项与子菜单联系起来
21      gtk_widget_show(Menu);
22   }
23   int main(int argc,char *argv[ ])
24   {
25      GtkWidget *window;                            //定义窗体
26      GtkWidget *MenuBar;                           //定义菜单条
27      GtkWidget *box;                               //定义组合框
28      GtkWidget *MenuItemFile;
29      gtk_init(&argc,&argv);
30      window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
31      gtk_widget_set_usize(window,400,200);         //设置窗体大小
32      gtk_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
33      box = gtk_vbox_new(FALSE,0);                            //创建纵向组合框
34      gtk_container_add(GTK_CONTAINER(window),box);           //把组合框加入窗体
35      MenuBar = gtk_menu_bar_new();                           //创建菜单条
36      gtk_box_pack_start(GTK_BOX(box),MenuBar,FALSE,TRUE,0);        //把菜单条加入组合框
37      MenuItemFile=CreateMenuItem(MenuBar,"File");                  //调用创建菜单项函数
38      CreateMenu(MenuItemFile);                                     //调用创建子菜单函数
39      CreateMenuItem(MenuBar,"Edit");
40      CreateMenuItem(MenuBar,"View");
41      CreateMenuItem(MenuBar,"Insert");
42      CreateMenuItem(MenuBar,"Tool");
43      gtk_widget_show(box);
44      gtk_widget_show(MenuBar);
45      gtk_widget_show(window);
46      gtk_main();
47   }
```

12.5.2 菜单的信号处理

对一个应用程序而言，建立完菜单之后，还需要对菜单的相应动作（信号）进行处理，也就是需要建立这些信号的回调函数，在这些回调函数中将对菜单栏中每个菜单项的动作进行处理，以实现对应的功能。

【例 12.23】使用菜单实现程序的退出

例 12.23 是一个对例 12.20 中显示的菜单中的 exit 子菜单进行处理的实例，exit 子菜单需要实现的功能是退出当前程序。

实例的应用代码如下：

```
1	#include <gtk/gtk.h>
2	
3	GtkWidget *CreateMenuItem(GtkWidget *MenuBar,char *test);
4	GtkWidget *CreateMenu(GtkWidget *MenuItem);
5	
6	int main(int argc,char *argv[ ])
7	{
8	   GtkWidget *window;                                    //定义窗体
9	   GtkWidget *MenuBar;                                   //定义菜单条
10	   GtkWidget *box;                                       //定义组合框
11	   GtkWidget *MenuItemFile;                              //定义文件子菜单
12	   gtk_init(&argc,&argv);
13	   window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
14	   gtk_widget_set_usize(window,400,200);                 //设置窗体大小
15	   g_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
16	   box = gtk_vbox_new(FALSE,0);                          //创建纵向组合框
17	   gtk_container_add(GTK_CONTAINER(window),box);         //把组合框加入窗体
18	   MenuBar=gtk_menu_bar_new();                           //创建菜单条
19	   gtk_box_pack_start(GTK_BOX(box),MenuBar,FALSE,TRUE,0); //把菜单条加入组合框
20	   MenuItemFile = CreateMenuItem(MenuBar,"Fiel");        //调用创建菜单项函数*
21	   CreateMenu(MenuItemFile);                             /*调用创建子菜单函数*/
22	   CreateMenuItem(MenuBar,"Edit");
23	   CreateMenuItem(MenuBar,"View");
24	   CreateMenuItem(MenuBar,"Insert");
25	   CreateMenuItem(MenuBar,"Tool");
26	   gtk_widget_show(box);
27	   gtk_widget_show(MenuBar);
28	   gtk_widget_show(window);
29	   gtk_main();
30	}
31	
32	GtkWidget *CreateMenuItem(GtkWidget *MenuBar,char *test)
33	{
34	   GtkWidget *MenuItem;
35	   MenuItem = gtk_menu_item_new_with_label(test);        /*创建菜单项*/
36	   gtk_menu_shell_append(GTK_MENU_SHELL(MenuBar),MenuItem);
```

```
37      /*把菜单项加入菜单条，注意我们使用 gtk_menu_shell_append 是为了程序的方便*/
38      gtk_widget_show(MenuItem);
39      return MenuItem;
40    }
41
42    GtkWidget *CreateMenu(GtkWidget *MenuItem)
43    {
44      GtkWidget *Menu                                    ;/*定义子菜单*/
45      GtkWidget *Exit;                                   /*定义 exit 子菜单项*/
46      Menu = gtk_menu_new();                             /*创建子菜单*/
47      CreateMenuItem(Menu,"New");                        /*调用创建菜单项函数*/
48      CreateMenuItem(Menu,"Open");
49      CreateMenuItem(Menu,"Save");
50      Exit = CreateMenuItem(Menu,"Exit");
51      g_signal_connect(GTK_OBJECT(Exit),"activate",G_CALLBACK(gtk_main_quit),NULL);
52      gtk_menu_item_set_submenu(GTK_MENU_ITEM(MenuItem),Menu);
         /*把父菜单项与子菜单联系起来*/
53      gtk_widget_show(Menu);
54    }
```

12.5.3 工具栏

工具条通常和菜单配合起来以便为用户提供常用的快捷命令，GTK+提供了一系列函数用于对工具条进行操作，对这些函数说明如下。

- gtk_toolbar_new 函数：用于创建一个新的工具条，函数没有参数，返回值是一个指向新建工具条的指针，这个新建的工具条和菜单栏类似，也只是一个容器，上面并没有按钮，在其上应该添加一些带图标指示的按钮。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
Gtk_Widget* gtk_toolbar_new(void);
```

- gtk_toolbar_append_item 函数：创建一个带图标的按钮，其中参数 toolbar 为指向待添加按钮的工具条，参数 text 为按钮所显示的正文，tooltip_text 为工具提示正文，参数 tooltip_private_text 是一个按键查询值，参数 widget 为按键对应的图标，需要使用原图或者图片构件，可以使用函数 gtk_image_new_from_file 来建立一个图片构件，函数 callback 用于指定按钮对应的回调函数，参数 userdata 为传递给回调函数的附加参数，函数的返回值是一个指向按钮的指针。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
Gtk_Widget* gtk_toolbar_append_item(GtkToolbar *toolbar, const char *text,const char *tooltip_text, const
char *tooltip_private_text, GtkWidget *widget, GtkSignalFunc callback, gpointer userdata);
```

- gtk_image_new_from_file 函数：用于建立一个图片构件以供工具栏的按钮使用，其中参数 file 为用于生成图片构件的文件，函数的返回值为生成图片构件。对其标准调用格式说明如下：

```
#include <gtk/gtk.h>
Gtk_Widget* gtk_image_new_from_file(char *file);
```

【例 12.24】工具栏的应用实例

例 12.24 是一个工具栏的应用实例，其建立了一个如图 12.22 所示的工具栏，当单击工具栏中各个按钮时会在终端中输出对应的字符串。在工具栏单击事件的回调函数 ButtonEvent 中调用了 g_print 函数，用于输出当前被单击的工具按钮对应的提示字符串。



图 12.22　工具栏的应用实例

实例的应用代码如下：

```
1       #include   <gtk/gtk.h>
2
3       void ButtonEvent(GtkWidget *widget,gpointer *data);
4
5       int main(int argc,char *argv[ ])
6       {
7          GtkWidget *window;
8          GtkWidget *box;     //定义组合盒
9          GtkWidget *toolbar;//定义工具条
10         GtkWidget *image; //定义图片构件
11         gtk_init(&argc,&argv);
12         window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
13         gtk_widget_set_usize(window,1000,400);
14         gtk_signal_connect(GTK_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
15         box = gtk_vbox_new(FALSE,0); //创建组合盒
16         toolbar = gtk_toolbar_new(); //创建工具条
17         gtk_box_pack_start(GTK_BOX(box),toolbar,FALSE,TRUE,5);/*把工具条加入组合盒*/
18         image = gtk_image_new_from_file("1.ico");
19         gtk_toolbar_append_item(GTK_TOOLBAR(toolbar),"office",
           "office",NULL,image,(GtkSignalFunc)ButtonEvent, "office");
20         //创建工具条里的按钮
21         image=gtk_image_new_from_file("2.ico");
22         gtk_toolbar_append_item(GTK_TOOLBAR(toolbar),"bitcomet",
           "bitcomet",NULL,image,(GtkSignalFunc)ButtonEvent, "bitcomet");
23         image = gtk_image_new_from_file("3.ico");
24         gtk_toolbar_append_item(GTK_TOOLBAR(toolbar),"blender",
           "blender",NULL,image,(GtkSignalFunc)ButtonEvent, "blender");
25         image = gtk_image_new_from_file("4.ico");
            MenuItemFile = CreateMenuItem(MenuBar,"Fiel");       //调用创建菜单项函数*
26         gtk_toolbar_append_item(GTK_TOOLBAR(toolbar),"coeur",
           "coeur",NULL,image,(GtkSignalFunc)ButtonEvent, "coeur");
27         image = gtk_image_new_from_file("5.ico");
28         gtk_toolbar_append_item(GTK_TOOLBAR(toolbar),"PS",
           "PS",NULL,image,(GtkSignalFunc)ButtonEvent, "PS");
29         gtk_container_add(GTK_CONTAINER(window),box);
30         gtk_widget_show(toolbar);
```

```
31          gtk_widget_show(box);
32          gtk_widget_show(window);
33          gtk_main();
34      }
35
36      void ButtonEvent(GtkWidget *widget,gpointer *data) /*回调函数*/
37      {
38          g_print("%s\n",data);
39      }
```

编译运行以上代码，单击各个按钮，可以在终端中看到对应的输出。

```
alloy@ubuntu:~/linuxc/chapter12$ ./exam1228toolbar
office
bitcomet
blender
coeur
PS
```

12.6　使用 Glade 界面设计师

从前面的章节中可以看到要想设计一个完整的拥有 Linux 图形界面的应用程序是非常麻烦的，如果需要设计一个大型的图形界面应用软件，可以使用 Glade 界面设计师。

Glade 界面设计师是 Linux 系统中设计 GTK+程序界面的所见即所得工具（类似于微软的 VS）GNOME 桌面环境的子项目，其是符合 GPL 协议的开源软件，用户可以通过直接向界面中的画布中直接添加构件的方式来建造自己应用程序的图形界面，该图形界面可以以 XML 格式保存，所以界面和对应的代码是完全独立的。

Glade 界面设计师的运行界面如图 12.23 所示，在其中放置了一个第 12.4.4 小节中介绍的日历构件，其具体使用方法在此不做详细介绍，有兴趣的读者可以自行查阅相应的资料。

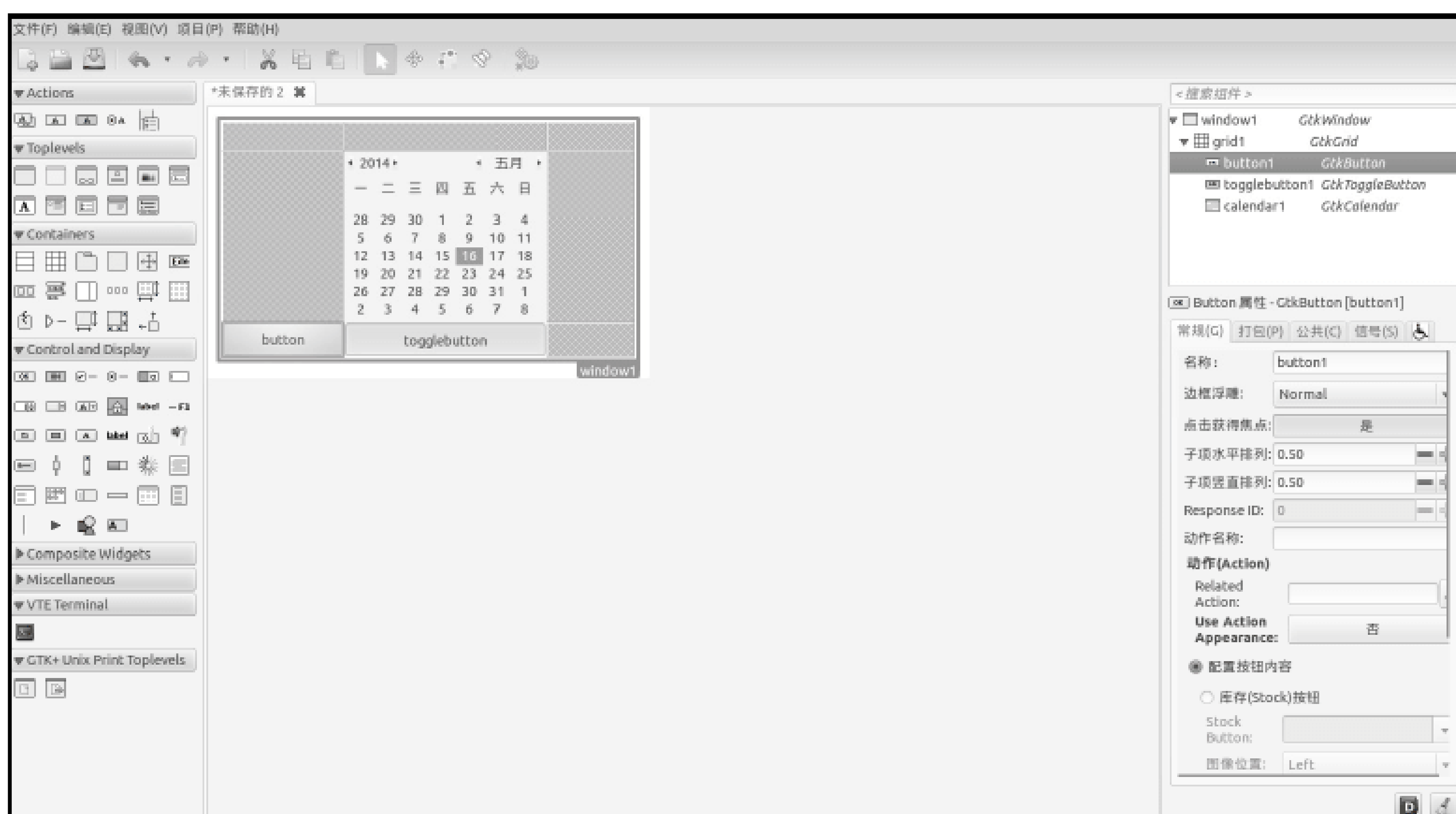


图 12.23　Glade 图形界面设计师的运行界面

12.7 本章习题

1. 编写一个程序，创建一个最简单的 GTK+窗口，并为窗口设置标题。

2. 编写一个程序，创建如图 12.24 所示的窗口，其中文本框内可输入的最多字符数为 20，当单击“提交”按钮时在终端输出对应的输入字符串。



图 12.24 “文本框的使用”窗口

3. 编写一个程序，创建如图 12.25 所示的窗口，单击“计数”按钮时显示当前单击按钮的次数。



图 12.25 “信号与事件”窗口

4. 编写一个程序，创建一个如图 12.26 所示的窗口，其中 3 个单选按钮和 2 个复选框，3 个单选按钮的选项分别为 Chinese、Math 和 English，2 个复选框的选项为 Teacher 和 Student。

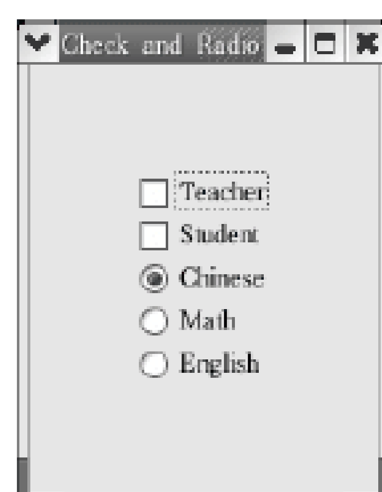


图 12.26 单选按钮和复选框

5. 创建一个带菜单和快捷工具栏的窗体。

第 13 章　Linux 的 C 语言编程实战

本章通过两个不同类比的应用实例介绍了在 Linux 中进行实际 C 语言编程的方法，涉及以下内容：

- 实时风力数据采集仪 PC 机端软件设计的应用实例。
- 俄罗斯方块的应用实例。

13.1　实时风力数据采集仪 PC 机端软件设计

实时风力数据采集仪是用于采集当前实时风力数据的设备，但是该设备需要通过串口和 PC 机连接才能将当前数据以文件的形式记录下来，本小节将介绍其 PC 端软件的设计方法。

13.1.1　实时风力数据采集仪 PC 机端软件的需求分析

实时风力数据采集仪的结构如图 13.1 所示，采集仪通过串口和 PC 机连接，PC 机端软件每隔 1 秒向采集仪发送一个字符串，当采集仪接收到这个字符串之后将当前风力数据以字符串的形式反馈给 PC 机端软件，PC 机端软件将该数据以文本文件的形式保存。



图 13.1　实时风力数据采集仪结构

对实时风力数据采集仪的 PC 机端软件和采集的数据交互方法说明如下：

- PC 机端软件每隔 1 秒通过串口定时向采集仪硬件发送字符串“Plz Send Data”，采集仪收到该字符串之后向 PC 机端软件回送当前风力数据，风力数据的格式为小数点前 1 位整数+小数点后 2 位的字符串，如 7.23，单位为米/秒。
- PC 端软件接收到风力数据之后将其和当前的时间信息连接形成一个完整的字符串，其中时间信息格式为“Thu Feb 21 20:09:25 2013”，风力数据的字符串格式为接收到的数据，然后将该完整的字符串写入文本文件，这个完整的字符串格式如下：

```
时间                          风力数据
Thu Feb 21 20:09:25 2014        7.23
Thu Feb 21 20:09:26 2014        7.22
```

- 当启动 PC 端软件的时候，其会使用启动时的时间作为文件名来创建一个新的文本文件，用于记录风力数据；当 PC 软件运行时，其会在屏幕上显示对应的提示字符串，并且当用户输入“quit”的时候退出 PC 端软件。

13.1.2 Linux 下的串口编程基础

实时风力数据采集仪的软件可以分为两个部分：串口数据的发送和接收；定时文件记录，后者在第 3.3 节中已经进行了详细介绍，本小节将介绍串口编程的基础知识。

1. 串口的硬件描述

串口是 Linux 系统中最常用的数据输入输出通道之一，尤其是在和嵌入式系统进行数据交互的时候，其利用一条传输线将数据以比特位为单位顺序传送。特点是通信线路简单，利用简单的线缆就可实现通信、降低成本，适用于传输距离长且传输速度较慢的通信。

PC 的串口通常使用了 MAX232 芯片作为接口器件，其内含两套电源变换电路，其中一个升压泵将 5V 电源提升到 10V，而另外一个反相器则提供-10V 的相关信号。该芯片是符合 RS-232-C 标准的通信芯片，一个标准的 RS-232-C 接口包括一个 25 针的 D 型插座（有公型和母型两种），包括主信道和辅助信道两个通信信道，且主信道的通信速率高于辅助信道。在实际使用中，常常只使用一个主信道，此时 RS-232-C 接口只需 9 根连接线，使用一个简化为 9 针的 D 型插座，同样也分为公型和母型，表 13.1 是 RS-232-C 接口的引脚定义。

表 13.1 RS-232-C 接口的引脚定义

25 针接口	9 针接口	名称	方向	功能说明
2	3	TXD	输出	数据发送引脚
3	2	RXD	输入	数据接收引脚
4	7	RTS	输出	请求数据传送引脚
5	8	CTS	输入	清除数据传送引脚
6	6	DSR	输出	数据通信装置 DCE 准备就绪引脚
7	5	GND		信号地
8	1	DCD	输入	数据载波检测引脚
20	4	DTR	输出	数据终端设备 DTE 准备就绪引脚
22	9	RI	输入	振铃信号引脚

RS-232-C 标准推荐的最大物理传输距离为 15 米，其逻辑电平“0”为+3V~+25V，而逻辑电平“1”为-3V~-25V，较高的电平保证了信号传输不会因为衰减导致信号的丢失。

串口对应的设备文件虽然在本质上也是第 3 章中介绍过的文件，但是在具体的使用方法上还是有一些区别的，其操作主要可分为初始化串口、发送数据、接收数据、处理中断和设置波特率等几个部分。

2. 串口对应的文件和结构体

在 Linux 系统中，所有的设备文件一般都位于“/dev”下，其中串口 1 和串口 2 对应的设备名依次为“/dev/ttyS0”和“/dev/ttyS1”，而且 USB 转串口的设备名通常为“/dev/ttyUSB0”和“/dev/ttyUSB1”（因版本不同，该设备名也会有所不同），可以查看在“/dev”下的文件进行确认。在 Linux 下对设备文件的操作方法与对普通文件的操作方法相同，对串口的读写可以使用 read、

write 等函数进行，需要注意的是需要对串口的其他参数另做配置

在 termios.h 文件中定义了结构体 termios，用于对串口进行初始化和控制，其是在 POSIX 规范中定义的标准接口，表示终端设备（包括虚拟终端、串口等）。

```
#include<termios.h>
struct termios
{
        unsigned short   c_iflag;              /*  输入模式标志  */
        unsigned short   c_oflag;              /*  输出模式标志  */
        unsigned short   c_cflag;              /*  控制模式标志*/
        unsigned short   c_lflag;              /*  本地模式标志  */
        unsigned char    c_line;               /*  线路规程  */
        unsigned char    c_cc[NCC];            /*  控制特性  */
        speed_t          c_ispeed;             /*  输入速度  */
        speed_t          c_ospeed;             /*  输出速度  */
};
```

3. 终端设备和其工作模式

串口是一种终端设备，一般通过终端编程接口对其进行配置和控制。终端有 3 种工作模式，分别为规范模式（canonical mode）、非规范模式（non-canonical mode）和原始模式（raw mode）。

通过在 termios 结构的 c_lflag 中设置 ICANNON 标志来定义终端是以规范模式（设置 ICANNON 标志）还是以非规范模式（清除 ICANNON 标志）工作，默认情况下为规范模式。

在规范模式下，所有的输入是基于行进行处理。在用户输入一个行结束符（回车符、EOF 等）之前，系统调用 read 函数时读不到用户输入的任何字符。除了 EOF 之外，行结束符（回车符等）与普通字符一样会被 read()函数读取到缓冲区之中。在规范模式中，行编辑是可行的，而且一次调用 read()函数最多只能读取一行数据。如果在 read 函数中被请求读取的数据字节数小于当前行可读取的字节数，则 read 函数只会读取被请求的字节数，剩下的字节下次再被读取。

在非规范模式下，所有的输入是即时有效的，不需要用户另外输入行结束符，而且不可进行行编辑。在非规范模式下，对参数 MIN（c_cc[VMIN]）和 TIME（c_cc[VTIME]）的设置决定 read 函数的调用方式，通常情况下有如下 4 种不同的情况。

- MIN = 0 和 TIME = 0：read 函数立即返回。若有可读数据，则读取数据并返回被读取的字节数，否则读取失败并返回 0。
- MIN > 0 和 TIME = 0：read 函数会被阻塞直到 MIN 个字节数据可被读取。
- MIN = 0 和 TIME > 0：只要有数据可读或者经过 TIME 个十分之一秒的时间，则 read 函数立即返回，返回值为被读取的字节数。如果超时并且未读到数据，则 read 函数返回 0。
- MIN > 0 和 TIME > 0：当有 MIN 个字节可读或者两个输入字符之间的时间间隔超过 TIME 个十分之一秒时，read 函数才返回。因为在输入第一个字符之后系统才会启动定时器，所以在这种情况下，read 函数至少读取一个字节之后才返回。

严格来说原始模式是一种特殊的非规范模式，在该模式下所有的输入数据以字节为单位被处理。在这个模式下，终端是不可回显的，而且所有特定的终端输入/输出控制处理不可用。通过调

用 cfmakeraw 函数可以将终端设置为原始模式，而且该函数对应的操作代码如下：

```
termios_p->c_iflag &= ~(IGNBRK | BRKINT | PARMRK | ISTRIP
                    | INLCR | IGNCR | ICRNL | IXON);
    termios_p->c_oflag &= ~OPOST;
    termios_p->c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);
    termios_p->c_cflag &= ~(CSIZE | PARENB);
    termios_p->c_cflag |= CS8;
```

4. 串口的设置常量

在对串口进行读写操作之前应该先对其进行设置，包括波特率、校验位和停止位等，在 termios 结构体中最重要的成员是 c_cflag，通过对它赋值，用户可以设置波特率、字符大小、数据位、停止位、奇偶校验位和硬软流控等，Linux 系统提供了一系列的常量用于对 c_cflag 进行操作，其中常用的常量说明如表 13.2 所示。

表 13.2　c_cflag 对应的操作常量

常量	说明
CBAUD	波特率的位掩码
B0	0 波特率（放弃 DTR）
B1800	1800 波特率
B2400	2400 波特率
B4800	4800 波特率
B9600	9600 波特率
B19200	19200 波特率
B38400	38400 波特率
B57600	57600 波特率
B115200	115200 波特率
EXTA	外部时钟率
EXTB	外部时钟率
CSIZE	数据位的位掩码
CS5	5 个数据位
CS6	6 个数据位
CS7	7 个数据位
CS8	8 个数据位
CSTOPB	2 个停止位（若不设，则是 1 个停止位）
CREAD	接收使能
PARENB PARODD	校验位使能 使用奇校验而不使用偶校验
HUPCL	最后关闭时挂线（放弃 DTR）
CLOCAL	本地连接（不改变端口所有者）
CRTSCTS	硬件流控


注　意

在实际使用中不能直接对 c_cflag 成员初始化，而是将其通过“与”、“或”操作使用其中的某些选项。

除了 c_cflag 之外，用户还可以通过对输入模式控制成员 c_iflag 的操作来控制对接收到的字符进行处理，Linux 同样提供了如表 13.3 所示的操作常量。

表 13.3　c_iflag 对应的操作常量

常量	说明
INPCK	奇偶校验使能
IGNPAR	忽略奇偶校验错误
PARMRK	奇偶校验错误掩码
ISTRIP	删除第 8 位比特
IXON	启动输出软件流控
IXOFF	启动输入软件流控
IXANY	输入任意字符可以重新启动输出（默认为输入起始字符才重启输出）
IGNBRK	忽略输入终止条件
BRKINT	当检测到输入终止条件时发送 SIGINT 信号
INLCR	将接收到的 NL（换行符）转换为 CR（回车符）
IGNCR	忽略接收到的 CR（回车符）
ICRNL	将接收到的 CR（回车符）转换为 NL（换行符）
IUCLC	将接收到的大写字符映射为小写字符
IMAXBEL	当输入队列满时响铃

c_oflag 用于对串口发送的字符进行控制，其对应的操作常量如表 13.4 所示。

表 13.4　c_oflag 对应的操作常量

常量	说明
OPOST	启用输出处理功能，如果不设置该标志，则其他标志都被忽略
OLCUC	将输出中的大写字符转换成小写字符
ONLCR	将输出中的换行符（‘\n’）转换成回车符（‘\r’）
ONOCR	如果当前列号为 0，则不输出回车符
OCRNL	将输出中的回车符（‘\r’）转换成换行符（‘\n’）
ONLRET	不输出回车符
OFILL	发送填充字符以提供延时
OFDEL	如果设置该标志，则表示填充字符为 DEL 字符，否则为 NULL 字符
NLDLY	换行延时掩码
CRDLY	回车延时掩码

（续表）

常量	说明
TABDLY	制表符延时掩码
BSDLY	水平退格符延时掩码
VTDLY	垂直退格符延时掩码
FFLDY	换页符延时掩码

c_lfag 分量用于控制串口的本地数据处理和工作模式，其对应的操作常量如表 13.5 所示。

表 13.5　c_lfag 对应的操作常量

常量	说明
ISIG	若收到信号字符（INTR、QUIT 等），则会产生相应的信号
ICANON	启用规范模式
ECHO	启用本地回显功能
ECHOE	若设置 ICANON，则允许退格操作
ECHOK	若设置 ICANON，则 KILL 字符会删除当前行
ECHONL	若设置 ICANON，则允许回显换行符
ECHOCTL	若设置 ECHO，则控制字符（制表符、换行符等）会显示成“^X”，其中 X 的 ASCII 码等于给相应控制字符的 ASCII 码加上 0x40，例如：退格字符（0x08）会显示为“^H”（‘H’的 ASCII 码为 0x48）
ECHOPRT	若设置 ICANON 和 IECHO，则删除字符（退格符等）和被删除的字符都会被显示
ECHOKE	若设置 ICANON，则允许回显在 ECHOE 和 ECHOPRT 中设定的 KILL 字符
NOFLSH	在通常情况下，当接收到 INTR、QUIT 和 SUSP 控制字符时，会清空输入和输出队列。如果设置该标志，则所有的队列不会被清空
TOSTOP	若一个后台进程试图向它的控制终端进行写操作，则系统向该后台进程的进程组发送 SIGTTOU 信号，该信号通常终止进程的执行
IEXTEN	启用输入处理功能

cc 分量用于对串口的特殊操作进行控制，其对应的操作常量如表 13.6 所示。

表 13.6　cc 对应的操作常量

常量	说明
VINTR	中断控制字符，对应键为 Ctrl+C
VQUIT	退出操作符，对应键为 Ctrl+Z
VERASE	删除操作符，对应键为 Backspace（BS）
VKILL	删除行符，对应键为 Ctrl+U
VEOF	文件结尾符，对应键为 Ctrl+D
VEOL	附加行结尾符，对应键为 Carriage return（CR）
VEOL2	第二行结尾符，对应键为 Line feed（LF）
VMIN	指定最少读取的字符数
VTIME	指定读取的每个字符之间的超时时间

5. 串口的操作流程

在 Linux 下对串口的操作流程如下。

01 调用函数 tcgetattr 来测试串口是否可用，并且保存串口的原始设置，对该函数的标准调用格式说明如下，如果调用成功，则函数返回值为 0，同时得到 fd 指向的终端配置参数，并将它们保存于 termios 结构变量 old_cfg 中；如果调用失败，则函数返回值为-1。

```
#include<termios.h>
int tcgetattr(int fd, struct termios *termios_p);
```

02 通过对 c_cflag 分量的设置（常量为 CLOCAL|CREAD）实现本地连接和接收使能，然后调用 cfmakeraw 函数把串口设置为原始模式。

03 调用 cfsetispeed 和 cfsetospeed 来对串口的输入波特率和输出波特率进行设置，对这两个函数的标准调用格式说明如下，通常来说输入波特率和输出波特率需要设置为相同。

```
#include<termios.h>
int cfsetospeed(struct termios *termptr, speed_t speed);
int cfsetispeed(struct termios *termptr, speed_t speed);
```



注 意

其中 struct termios *termptr 是指向 termios 结构的指针；而 speed_t speed 是需要设置的波特率，如果调用函数成功，则返回 0，若调用该函数失败，则返回-1。

04 通过对 c_cflag 的位掩码设置来设置字符的大小，即每个字节中存在几位数据。

05 通过对 c_cflag 和 c_iflag 位的操作来设置奇偶校验位，其对应的操作代码如下：

```
.c_cflag |= (PARODD | PARENB);
.c_iflag |= INPCK;
//以上是奇校验的操作代码
.c_cflag |= PARENB;
.c_cflag &= ~PARODD;     //若清除偶校验标志，则配置为奇校验
.c_iflag |= INPCK;
//以上是偶校验的操作代码
```

06 通过对 c_cflag 分量的操作来设置停止位，当停止位为 1 位的时候清除 c_cflag 中的 CSTOPB，当停止位为 2 位的时候激活 CSTOPB，其对应的操作代码如下：

```
.c_cflag &=   ~CSTOPB;     /* 将停止位设置为一个比特 */
.c_cflag |=   CSTOPB;      /* 将停止位设置为两个比特 */
```

07 通过 c_cc 分量的设置来修改字符缓冲区和等待时间，如果缓冲区的大小设置为 0，则在串口读到一个字节之后立即返回，其对应的操作代码如下：

```
.c_cc[VTIME] = 0;
.c_cc[VMIN] = 0;
```

08 使用 tcdrain 系列函数清理当前串口的缓冲区，对 tcdrain 系列函数的标准调用格式说明如

下：

```
#include<termios.h>
int tcdrain(int fd);
int tcflow(int fd, int action) ;
int tcflush(int fd, int queue_selector);
```



tcdrain 函数用于使程序阻塞，直到输出缓冲区的数据全部发送完毕，其中 fd 参数为串口的文件描述符。

tcflow 函数用于暂停或重新开始输出，其中 fd 参数为串口的文件描述符，action 参数有如下 4 种可能。

- TCOOF：输出被挂起。
- TCCON：重新启动以前被挂起的操作。
- TCIOFF：系统发送一个 STOP 字符，以使终端设备暂停发送数据。
- TCION：系统发送一个 START 字符，使得终端设备恢复发送数据。

tcflush 函数用于清空输入/输出缓冲区，其参数 queue_selector 用于选择对缓冲区的处理方式，有如下 3 种可能的取值：

- TCIFLUSH：对接收到而未被读取的数据进行清空处理。
- TCOFLUSH：对尚未传送成功的输出数据进行清空处理。
- TCIOFLUSH：包括前两种功能，即对尚未处理的输入输出数据进行清空处理。

这 3 个函数如果调用成功均返回 0，若调用失败，则返回-1。

09 调用 tcsetattr 函数激活当前的串口配置，对该函数的标准调用格式说明如下：

```
#include<termios.h>
tcsetattr(int fd, int optional_actions, const struct termios *termios_p);
```



其中参数 termios_p 是 termios 类型的新配置变量，参数 optional_actions 可能的取值有以下 3 种。

- TCSANOW：配置的修改立即生效。
- TCSADRAIN：配置的修改在所有写入 fd 的输出都传输完毕之后生效。
- TCSAFLUSH：所有已接受但未读入的输入都将在修改生效之前被丢弃。

此时可以按照文件的操作方式使用 open 函数、write 函数和 read 函数对串口进行操作，这些函数和普通的读写操作略有差别。

例 13.1 给出了一个完整的对串口进行初始化操作的函数 set_com_config 的实例代码，其提供了如下参数用于对串口进行设置，如果初始化操作成功，则函数返回值为 0。

- baud_rate：设置串口的波特率，有 2400（bps）、4800、9600、19200、38400 和 115200

共 6 种选择。

- data_bits：设置串口的数据位，有 7 位和 8 位两种选择。
- parity：设置串口奇偶校验位，有 “N”、“O”、“E”、和 “S” 4 种选择。
- stop_bits：设置串口的停止位，有 “1” 和 “2” 两种选择。

【例 13.1】串口初始化配置函数 set_com_config

实例的应用代码如下：

```
1   #include <termios.h>
2
3   int set_com_config(int fd,int baud_rate,
4                          int data_bits, char parity, int stop_bits)
5   {
6       struct termios new_cfg,old_cfg;
7       int speed;
8
9       /*保存并测试现有串口参数设置，如果串口号等出错，会有相关的出错信息*/
10      if  (tcgetattr(fd, &old_cfg)  !=  0)
11      {
12          perror("tcgetattr");
13          return -1;
14      }
15      /*  设置字符大小*/
16      new_cfg = old_cfg;
17      cfmakeraw(&new_cfg); /*  配置为原始模式  */
18      new_cfg.c_cflag &= ~CSIZE;
19      /*设置波特率*/
20      switch (baud_rate)
21      {
22          case 2400:
23          {
24              speed = B2400;
25          }
26          break;
27          case 4800:
28          {
29              speed = B4800;
30          }
31          break;
32          case 9600:
33         {
34              speed = B9600;
35          }
36          break;
37          case 19200:
38          {
39             speed = B19200;
40          }
```

```
41                  break;
42                  case 38400:
43                  {
44                          speed = B38400;
45                  }
46                  break;
47
48                  default:
49                  case 115200:
50                  {
51                          speed = B115200;
52                  }
53                  break;
54          }
55          cfsetispeed(&new_cfg, speed);
56          cfsetospeed(&new_cfg, speed);
57
58          /*设置停止位*/
59          switch (data_bits)
60          {
61                  case 7:
62                  {
63                              new_cfg.c_cflag |= CS7;
64                  }
65                  break;
66
67                  default:
68                  case 8:
69                  {
70                              new_cfg.c_cflag |= CS8;
71                  }
72                  break;
73          }
74          /*设置奇偶校验位*/
75          switch (parity)
76          {
77                  default:
78                  case 'n':
79                  case 'N':
80                  {
81                          new_cfg.c_cflag &= ~PARENB;
82                          new_cfg.c_iflag &= ~INPCK;
83                  }
84                  break;
85
86                  case 'o':
87                  case 'O':
88                  {
89                          new_cfg.c_cflag |= (PARODD | PARENB);
```

```
90                      new_cfg.c_iflag |= INPCK;
91                  }
92                  break;
93
94                  case 'e':
95                  case 'E':
96                  {
97                      new_cfg.c_cflag |= PARENB;
98                      new_cfg.c_cflag &= ~PARODD;
99                      new_cfg.c_iflag |= INPCK;
100                 }
101                 break;
102
103                 case 's':   /*as no parity*/
104                 case 'S':
105                 {
106                 new_cfg.c_cflag &= ~PARENB;
107                 new_cfg.c_cflag &= ~CSTOPB;
108                 }
109                 break;
110             }
111
112             /*设置停止位*/
113             switch (stop_bits)
114             {
115                 default:
116                 case 1:
117                 {
118                     new_cfg.c_cflag &=   ~CSTOPB;
119                 }
120                 break;
121
122                 case 2:
123                 {
124                     new_cfg.c_cflag |= CSTOPB;
125                 }
126             }
127
128             /*设置等待时间和最小接收字符*/
129             new_cfg.c_cc[VTIME]   = 0;
130             new_cfg.c_cc[VMIN] = 1;
131
132             /*处理未接收字符*/
133             tcflush(fd, TCIFLUSH);
134             /*激活新配置*/
135             if ((tcsetattr(fd, TCSANOW, &new_cfg)) != 0)
136             {
137                 perror("tcsetattr");
138                 return -1;
```

```
139         }
140         return 0;
141   }
```

打开串口，和文件一样，在使用串口之前必须使用 open 函数来打开串口，其标准调用格式如下，其中使用了 O_NOCTTY 和 O_NDELAY 参数：

```
fd = open( "/dev/ttyS0", O_RDWR|O_NOCTTY|O_NDELAY);
```

- O_NOCTTY 标志用于通知 Linux 系统，该参数不会使打开的文件成为这个进程的控制终端。如果没有指定这个标志，那么任何一个输入（诸如键盘中止信号等）都将会影响用户的进程。
- O_NDELAY 标志通知 Linux 系统，这个程序不关心 DCD 信号线所处的状态（端口的另一端是否激活或者停止）。如果用户指定了这个标志，则进程将会一直处在睡眠状态，直到 DCD 信号线被激活。

打开串口之后应该调用 fcntl 函数来将串口的状态设置为阻塞，以等待数据输入，其标准调用格式如下：

```
fcntl(fd, F_SETFL, 0);
```

如果对已经打开的串口状态不放心，可以使用 isatty 函数来测试串口是否正确打开，对其标准调用格式说明如下，参数 STDIN_FINENO 为串口对应的文件描述符，如果串口打开成功会返回 0，否则返回-1。

```
#include<termios.h>
isatty(STDIN_FILENO);
```

此时一个串口就已经成功打开了，接下来就可以对这个串口进行读和写操作，例 13.2 给出了一个完整的串口打开操作函数 open_port 的实例代码，其充分考虑到了打开串口过程中的各种状态，其中参数 com_port 为串口的编号，函数的返回值为串口对应的文件描述符。

【例 13.2】串口打开操作函数 open_port

实例的应用代码如下：

```
 1  #include<termios.h>
 2  #define MAX_COM_NUM      32   //最大串口数目
 3
 4  int open_port(int com_port)
 5  {
 6          int fd;
 7  #if (COM_TYPE == GNR_COM)   /*  使用普通串口  */
 8          char *dev[] = {"/dev/ttyS0", "/dev/ttyS1", "/dev/ttyS2"};
 9  #else /*  使用 USB 转串口  */
10          char *dev[] = {"/dev/ttyUSB0", "/dev/ttyUSB1", "/dev/ttyUSB2"};
11  #endif
12          if ((com_port < 0) || (com_port > MAX_COM_NUM))
```

```
13          {
14                  return -1;
15          }
16          /*  打开串口  */
17          fd = open(dev[com_port - 1], O_RDWR|O_NOCTTY|O_NDELAY);
18          if (fd < 0)
19          {
20                      perror("open serial port");
21                      return(-1);
22          }
23
24          /*恢复串口为阻塞状态*/
25          if (fcntl(fd, F_SETFL, 0) < 0)
26          {
27                  perror("fcntl F_SETFL\n");
28          }
29
30          /*测试是否为终端设备*/
31          if (isatty(STDIN_FILENO) == 0)
32          {
33                  perror("standard input is not a terminal device");
34          }
35          return fd;
36  }
```

此时可以和对普通文件操作一样使用 read 函数和 write 函数对串口进行读写操作。

13.1.3 实时风力数据采集仪 PC 机端软件的代码设计

实时风力数据采集仪的 PC 端软件的工作流程如图 13.2 所示，对其步骤描述如下：

01 系统初始化，调用串口初始化配置函数，对串口进行初始化操作，调用串口打开函数打开串口。

02 获取当前的时间信息，调用文件处理函数创建一个新的文件，用于记录实时风力数据，使用当前时间信息字符串作为文件名。

03 在屏幕上输出工作信息，通过串口发送字符串命令“Plz Send Data”。

04 等待串口返回风力数据。

05 获取当前时间信息，和风力数据进行拼接形成记录字符串。

06 将记录字符串写入记录文件。

07 检查用户是否输入了“quit”字符串，如果是，则关闭文件，退出应用程序，否则等待 1 秒后回到步骤 3。

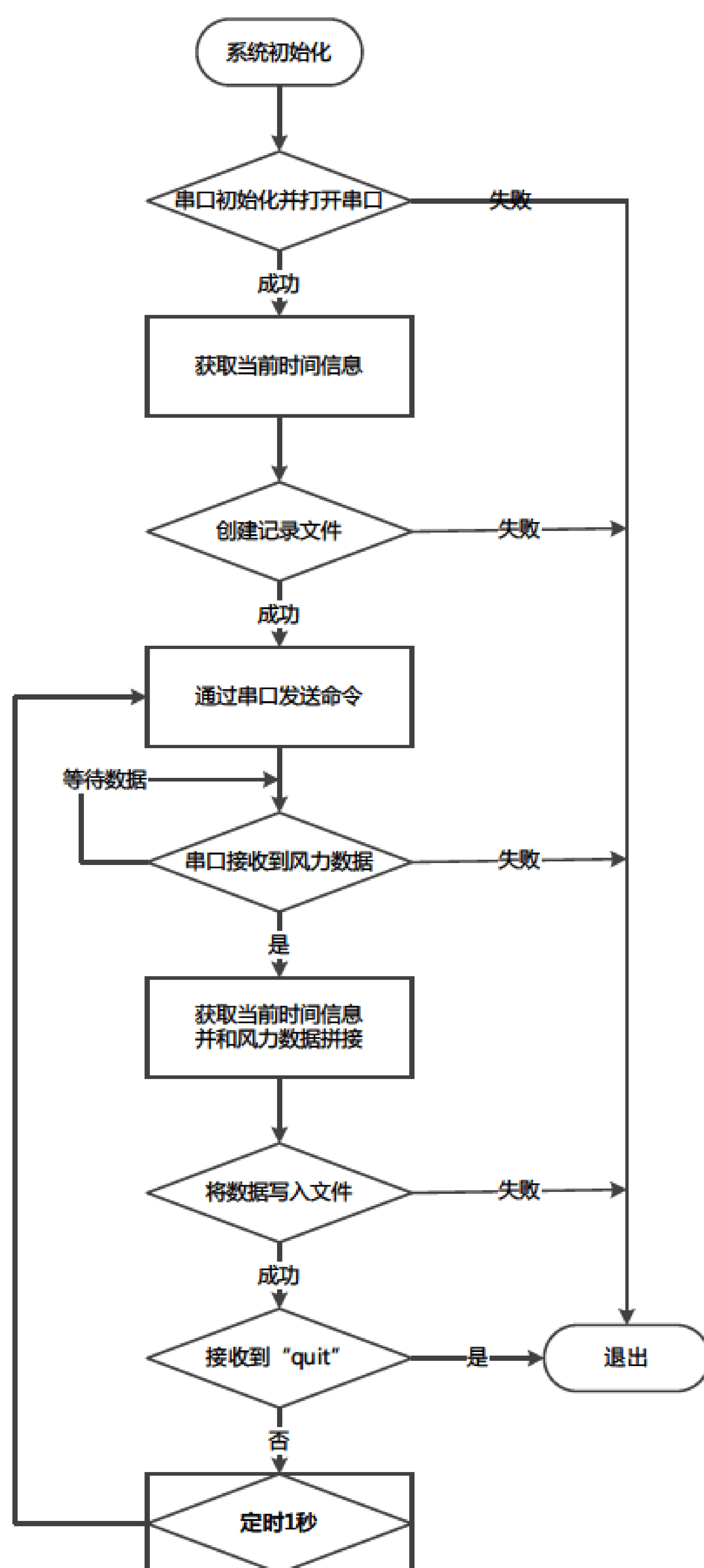


图 13.2　实时风力数据采集仪 PC 机端软件

PC 机端软件的应用代码如例 13.3 所示，其中调用了例 13.1 和例 13.2 的串口操作函数对串口进行操作，还使用了第 3 章介绍的文件编程方法来对记录文件进行处理，其中对于时间和字符串的处理方法可以参考例 3.14~例 3.18。

【例 13.3】实时风力数据采集仪 PC 端软件的应用代码

实例的应用代码如下：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <sys/types.h>
5   #include <sys/stat.h>
6   #include <errno.h>
7   #include <unistd.h>
8   #include <sys/time.h>
9   #include <fcntl.h>
10  #include <termios.h>
11
```

```
12  #define INPUTBUFFSIZE    20                  //风力数据输入缓冲区大小
13  #define QUITBUFFSIZE     20                  //用户输入缓冲区大小
14  #define WRITEBUFFSIZE   50                   //写文件缓冲区大小
15  #define MAX_COM_NUM      32                  //最大串口数目
16  #define HOST_COM_PORT   1                    //使用 1 号串口
17
18  int set_com_config(int fd,int baud_rate,int data_bits,char parity, int stop_bits)
19  {
20    //省略了具体代码，参考例 13.1
21  }
22
23  int open_port(int com_port)
24  {
25    //省略了具体代码，参考例 13.2
26  }
27
28  //以下是主函数的内容
29  int main(int argc,char *argv[])
30  {
31              int sfd;                                   //串口文件描述符
32              int recondfd;                              //记录文件描述符
33              char sendbuff[] = "Plz Send Data";         //发送字符串缓冲区
34              char inputbuff[INPUTBUFFSIZE];             //接收字符缓冲区
35              char quitbuff[QUITBUFFSIZE];               //用户输入缓冲区
36              char writebuff[WRITEBUFFSIZE];             //风力数据写入文件缓冲区
37              char enterbuf[3]="\r\n";                   //回车换行
38              int temp,seektemp;                         //偏移量计算中间量
39              struct timezone timez;
40              time_t timetemp;                           //时间结构体变量
41              int j = 0;
42              int writeCounter = 0;                      //写入计数器
43
44          if((sfd = open_port(HOST_COM_PORT)) < 0)       /*  打开串口  */
45          {
46                perror("open_port");                     //打开串口失败
47                return 1;
48          }
49          if(set_com_config(sfd, 115200, 8, 'N', 1) < 0)  /*  配置串口  */
50          {
51                perror("set_com_config");                //配置串口失败
52                return 1;
53          }
54        recondfd = get_new_file;     //创建一个新的文件用于记录风力数据
55        do
56          {
57                printf("System is running(enter 'quit' to exit):");     //输出提示信息，如果输入 quit 则退出
58                write(sfd,sendbuff,strlen(sendbuff));                   //通过串口发送命令
59              //以下代码用于获取用户的输入
60                memset(quitbuff, 0, QUITBUFFSIZE);
```

```
61              fgets(quitbuff, QUITBUFFSIZE, stdin);          //从键盘读取用户的输入存放到 quitbuff 中
62              usleep(50);                                    //等待 50 毫秒确定串口数据已经反馈回来
63              memset(inputbuff, 0, INPUTBUFFSIZE);           //为接收字符串缓冲区分配空间
64              if (read(sfd, inputbuff, INPUTBUFFSIZE) > 0)//从串口读取数据并且在屏幕上输出
65              {
66                  printf("WindSpeed is: %s", inputbuff);
67              }
68              //以下是将数据写入记录文件的内容
69              time(&timetemp);                               //获得当前时间参数
70              sprintf(writebuff,"%s",ctime(&timetemp));      //将当前时间参数放入写缓冲区
71              stract(writebuff,inputbuff);                   //将时间数据和风力数据连接
72              stract(writebuff,enterbuf);                    //在时间风力数据的最后添加上回车换行
73              if(writeCounter == 0)                          //第一次写入
74              {
75                  temp = write(recondfd,writebuff,strlen(writebuff));      //写入数据
76                  seektemp = lseek(recondfd,0,SEEK_CUR);                   //获得当前的偏移量
77                  writeCounter++;                                          //写入计数器
78              }
79              else
80              {
81                  j = strlen(writebuff) * writeCounter;                    //获得偏移量
82                  seektemp = lseek(recondfd,j,SEEK_SET);
83                  temp = write(recondfd,writebuff,strlen(writebuff));
84                  writeCounter++;
85              }
86          } while(strncmp(quitbuff, "quit", 4));             //如果检查到用户输入 quit 则退出
87          close(sfd);                                        //关闭串口
88          close(recondfd);                                   //关闭记录文件
89          return 0;
90      }
```

13.1.4　实时风力数据采集仪 PC 机端软件的记录数据

当完成风力数据的采集之后可以使用“cat –n”命令来查看当前记录的风力数据，其显示内容如下：

```
1       Tue Sep 2 10:09:25 2014     1.02
2       Tue Sep 2 10:09:26 2014     1.02
3       Tue Sep 2 10:09:27 2014     1.01
4       Tue Sep 2 10:09:28 2014     1.02
5       Tue Sep 2 10:09:29 2014     1.04
6       Tue Sep 2 10:09:30 2014     1.04
```

13.2　俄罗斯方块游戏设计

俄罗斯方块是一款风靡全球的电视游戏机和掌上游戏机游戏，它由俄罗斯人阿列克谢・帕基特诺夫发明，故得此名，其基本规则是移动、旋转和摆放游戏自动输出的各种方块，使之排列成完整的一行或多行并且消除得分。由于上手简单、老少皆宜，从而家喻户晓，风靡世界。

13.2.1 俄罗斯方块的需求分析

俄罗斯方块这个游戏有相当多的不同“变身”，其中的细节规则可能有千差万别，但是都具有如下相同的基本规则。

- 提供一个用于摆放小型正方形的平面虚拟场地，其标准大小：行宽为 10，列高为 20，以每个小正方形为单位。
- 提供一组由 4 个小型正方形组成的规则图形，英文名称为 Tetromino，中文通称为方块。这些方块共有 7 种不同的类似，分别以 S、Z、L、J、I、O、T 这 7 个字母的形状来命名：I 表示一次最多消除四层；J（左右）表示最多消除三层，或消除二层；L 表示最多消除三层，或消除二层；O 表示消除一至二层；S（左右）表示最多二层，容易造成孔洞；Z（左右）表示最多二层，容易造成孔洞；T 表示最多二层。图 13.3 是这 7 种方块的形状示意。

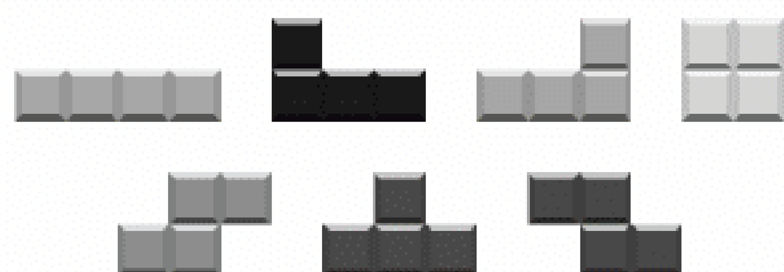


图 13.3　俄罗斯方块的 7 种图形

- 玩家可执行的操作有：以 90° 为单位旋转方块、以格子为单位左右移动方块、让方块加速落下。
- 当方块移到区域最下方或是到其他方块上无法移动时，就会固定在该处，而新的方块出现在区域上方开始落下。
- 当区域中某一列横向格子全部由方块填满，则该列会消失并成为玩家的得分，同时删除的列数越多，得分指数越高。
- 当固定的方块堆到区域最上方而无法消除层数时，则游戏结束。
- 一般来说，游戏还会提示下一个要落下的方块，熟练的玩家会计算到下一个方块，评估现在要如何进行。由于游戏能不断进行下去，对商业游戏不太理想，所以一般还会随着游戏的进行而加速，从而提高难度。
- 通过设计者预先设置的随机发生器不断地输出单个方块到场地顶部，以一定的规则进行移动、旋转、下落和摆放，锁定并填充到场地中。如果每次摆放将场地的一行或多行完全填满，则组成这些行的所有小正方形将被消除，并且以此来换取一定的积分或者其他形式的奖励。而未被消除的方块会一直累积，并对后来的方块摆放造成各种影响。
- 如果未被消除的方块堆放的高度超过场地所规定的最大高度（并不一定是 20 或者玩家所能见到的高度），则游戏结束。

13.2.2 GTK+的图形设计进阶

俄罗斯方块游戏的工作原理是在显示屏幕上绘制对应的方块图案，并且在游戏者的控制下对这些方块图案进行操控，当它们执行到符合规则的相应情况时，采取相应的动作，例如消除一行或者结束游戏；由于在设计中需要在当前屏幕上绘制对应的图形并且提供用户操作接口，所以需要使

用 GTK+库以完成在屏幕上绘制图形的工作。

在第 12 章中介绍了使用 GTK+在 Linux 中进行基础图形编程的方法，从其中可以知道 GTK+的图形设计可以分为两大部分：使用 GTK+在屏幕上绘制图形以及处理用户对图形操作所引起的事件。

1. GTK+的绘图

在 Linux 中实现屏幕上图形绘制的构件是绘图区构件，即 GdkWindow，其本质上是一个 X 窗口，它是一个空白的画布，可以在其上绘制需要的东西。GdkWindow 是 Xlib 窗口对象的封装。一个 GdkWindow 代表屏幕上的一个区域，可以显示或隐藏起来，也可以捕获 GdkWindow 接收到的事件，还可以在里面绘制图像、移动或调整图像的尺寸。GdkWindow 是以树状结构组织的，也就是说，每一个窗口都可以有子窗口。子窗口是相对于父窗口的位置定位的，当父窗口移动时，子窗口也会移动，子窗口不会在父窗口边界外的区域绘制。

绘图区构件利用如下函数创建：

```
GtkWidget* gtk_drawing_area_new (void);
```

创建绘图区构件后，利用如下函数设置构件的默认大小：

```
void gtk_drawing_area_size (GtkDrawingArea *darea, gint width, gint height);
```

当创建绘图区构件时应该注意，应用程序完全负责绘制其上的内容。如果应用程序窗口被遮住后暴露出来，则系统会发送一个曝光事件给应用程序，应用程序必须重绘先前被遮住的部分。曝光事件会告诉应用程序需要重绘部分的坐标、宽度和高度。

为了能正确的重绘，应用程序必须记住绘制在屏幕上的内容。如果窗口的一部分被清除了，需要一步步地重绘。显然这比较麻烦。解决的办法是使用一个 pixmap。可以将 pixmap 当作屏幕的一个缓冲区，当窗口需要重新绘制时，先在 pixmap 中绘制来代替直接向屏幕绘制，并且只绘制图像改变的部分，然后复制 pixmap 相应的部分到屏幕上即可。

可利用如下函数创建 pixmap：

```
GdkPixmap* gdk_pixmap_new (GdkWindow *window, gint width, gint height, gint depth);
```

window 参数用于设置一个 GDK 窗口，位图继承该窗口的所有属性。width 和 height 用于设置位图的大小。depth 用于设置颜色深度，如果 depth 设为-1，则它会自动匹配窗口的颜色深度。

一般情况下，在事件 configure_event 的回调函数中创建位图。这个事件会在窗口创建以及改变窗口大小时产生。

有了绘图区和颜色，要进行绘图还需要一种工具——“画笔”。图形上下文（GC，Graphics Context）就是用来设置“画笔”参数（例如前景色、后景色、字体等等）。

GC 与要绘图的可绘区相关联，不同的绘图区可以使用相同的 GC，也可以使用不同的 GC，如果要使用相同的 GC，则绘图区的颜色深度应该相同。创建 GC 可以使用 gdk_gc_new 函数，它的原型如下：

```
GdkGC *gdk_gc_new   (GdkDrawable   *drawable);
```

其参数是要应用该 GC 进行绘图的可绘区。创建 GC 后，下一步要定制 GC 参数。GC 参数是通过 GdkGCValues 结构体进行描述。结构体中包含了 GC 所有的特性。下面是 GdkGCValues 结构体的定义：

```
typedef struct _GdkGCValues                 GdkGCValues;
struct _GdkGCValues
{
  GdkColor              foreground;
  GdkColor              background;
  GdkFont      *font;
  GdkFunction           function;
  GdkFill         fill;
  GdkPixmap           *tile;
  GdkPixmap           *stipple;
  GdkPixmap           *clip_mask;
  GdkSubwindowMode   subwindow_mode;
  gint                  ts_x_origin;
  gint                  ts_y_origin;
  gint                  clip_x_origin;
  gint                  clip_y_origin;
  gint                  graphics_exposures;
  gint                  line_width;
  GdkLineStyle          line_style;
  GdkCapStyle           cap_style;
  GdkJoinStyle          join_style;
};
```

前景色（foreground 成员）是画线、圆或其他形状时的"画笔颜色"。背景色（background 成员）的用处依赖于特定的绘画操作。这些颜色必须是用 gdk_color_alloc 函数在当前颜色表中分配的。

在 GDK 以前的版本中，font 用来在绘制文本时指定字体，但是新的绘制文本的 GDK 程序都要求一个 GdkFont *参数，因此该参数可以忽略。

function 成员指定要画的像素点与可绘区上已有的像素点如何结合起来。有许多种可能的取值，但是只有两种是最常用的，如表 13.7 所示。

表 13.7　function 的常用取值

function 值	说明
GDK_COPY	缺省值，它忽略已存在的像素点（只是将新的像素点画在上面）
GDK_XOR	将旧的和新的像素点以一种可反转的方式结合起来，也就是说，如果执行两次 GDK_OR 操作，头一次绘图就会被第二次操作取消，GDK_XOR 通常用于"擦除"，可以恢复可绘区的原来内容

GdkGCValues 的 fill 成员用于决定如何使用 GdkGCValues 中的 tile 和 stipple 成员。其中 tile 成员是一个与目的可绘区深度相同的 pixmap 图片，它被反复复制到目的可绘区，将它们拼贴起来，第一次拼贴的原点是(ts_x_origin, ts_y_origin) 。而 stipple 成员是一个位图（深度为 1 的 pixmap），它也是从(ts_x_origin, ts_y_origin)开始拼贴的。fill 的可能取值如表 13.8 所示。

表 13.8　fill 的取值

fill 值	说明
GDK_SOLID	忽略 tile 和 stipple 成员，绘图形状是用前景色和背景色绘制的
GDK_TILED	绘图形状利用 tile 成员指定的 pixmap 图片绘制，而不是利用前景色和背景色。利用 GDK_TILED 模式绘画时会擦除可绘区上的任何内容，显示由 tile 成员指定的图片的拼贴图形
GDK_STIPPLED	利用 stipple 中定义的位绘制图形，也就是说，在 stipple 成员中未设置的位不会绘出
GDK_OPAQUE_STIPPLED	利用前景色绘制在 stipple 中设置的位，没有在 stipple 中设置的位利用背景色绘制

clip_mask 成员是可选的，它是一个位图，只有在这个位图中设置了的位才会画出。从 clip_ mask 到可绘区的映射是由 clip_x_origin 和 clip_y_origin 值决定的，这些定义了与 clip_mask 中的(0,0)对应的可绘区坐标。也可以设置一个剪裁矩形（最常用的，也是最有用的形式）或一个剪裁区域（区域就是在屏幕上的任意范围，典型情况是一个多边形或矩形列表）。若要关闭“剪裁”，可将剪裁的矩形、区域或剪裁屏蔽值设置为 NULL。

GC 的 subwindow_mode 只与可绘区是否为一个窗口有关。缺省设置是 GDK_CLIP_BY_CHILDREN，这意味着子窗口不会被在父窗口上的绘图所影响。

graphics_exposures 是一个布尔值，缺省值是 TRUE，它决定了 gdk_window_copy_area 函数是否产生曝光事件。

GC 的最后 4 个值决定了怎样画线。这些值用于画线，包括未填充多边形的边框以及弧线。

line_width 域决定了线的宽度（以像素计）。宽度为 0 的线称为一条“细线”，细线是一个像素宽的线，绘制得非常快（通常使用硬件加速），但是画的具体像素依赖于所使用的 X 服务器。为了保持一致性，最好使用宽度为 1 的线。

line_style 域可以是如表 13.9 所示的三种取值。

表 13.9　line_style 的取值

line_style 值	说明
GDK_LINE_SOLID	为缺省值，即一条实线
GDK_LINE_ON_OFF_DASH	用前景色画一条虚线，将虚线的 off（关闭）部分空着
GDK_LINE_DOUBLE_DASH	用前景色画一条虚线，但是虚线的 off（关闭）部分用背景色绘制

cap_style 用于决定 X 画线的端点（或虚线端点），它有如表 13.10 所示的 4 种可能取值。

表 13.10　cap_style 的取值

cap_style 值	说明
GDK_CAP_BUTT	缺省值，它意味着线的端点是正方形的
GDK_CAP_NOT_LAST	对应一个像素宽度的线，最后一个像素忽略不画。其他与 GDK_CAP_BUTT 相同

（续表）

cap_style 值	说明
GDK_CAP_ROUND	在线的端点画一个小弧线，由线的端点向两边延伸。弧线的中心是线的端点，半径是线宽的一半。对于一个像素宽的线，它没有什么效果（因为没有办法画一个像素宽的弧线）
GDK_CAP_PROJECTING	将线延伸，它对一个像素的线没有效果

join_style 参数将影响绘制多边形或在一个函数中绘制多条线时，各线之间如何连接。如果把线想象成一个细长的矩形，就很容易弄清楚线之间并不是平滑连接的。在连接的两个端点之间有一个凹槽。对这个凹槽有三种处理方法，也就是 join_style 的三种可能取值，如表 13.11 所示。

表 13.11 join_style 的取值

join_style 值	说明
GDK_JOIN_MITER	缺省值，在线交叉的地方画一个尖角
GDK_JOIN_ROUND	在交叉的凹槽处画一个弧线，画一个圆形的转角
GDK_JOIN_BEVEL	用最小的可能形状填充凹槽，画一个平坦的转角

GdkGCValues 中的各个参数一般通过函数 gdk_gc_set 进行设置，对其标准调用格式说明如下：

```
gdk_gc_set_参数名
```

也可以先设置 GC 的属性，然后再产生 GC，这时调用函数 gdk_gc_new_with_values，它的原型如下：

```
GdkGC *gdk_gc_new_with_values (GdkDrawable  *drawable, GdkGCValues *values,
GdkGCValuesMask     values_mask);
```

其中第 1 个参数是要使用该 GC 的可绘区，第 2 个参数是 GC 对应的参数，第 3 个参数是指明 GC 对应参数中哪些参数是有效的。

如果要删除 GC，可调用函数 gdk_gc_unref，对其标注调用格式说明如下，其参数是待删除的 GC。

```
void     gdk_gc_unref   (GdkGC *gc);
```

既然要制图，就少不了与颜色打交道，GDK 使用 GdkColor 结构存储颜色的 RGB 值和像素值。红、绿、蓝值是以 16 位无符号整数给出的，取值范围为 0~65535。下面是 GdkColor 的结构定义：

```
typedef struct _GdkColor GdkColor;
struct _GdkColor
{
gulong pixel;
gushort red;
gushort green;
gushort blue;
} ;
```

在利用一种颜色绘画时，必须保证像素值包含合适的值，并且保证颜色值在要使用的可绘区的颜色表中存在（可绘区是一个可以在上面绘画的窗口）。在 GDK 中可以通过调用 gdk_colormap_alloc_color 函数来填充像素值，并将颜色值添加到颜色表中，对其标准调用格式说明如下：

```
#include <gdk/gdk.h>
gboolean gdk_colormap_alloc_color(GdkColormap* colormap,GdkColor* color,gboolean writeable,
gboolean best_match)
```

第 1 个参数是要绘画可绘区的颜色表，第 2 个参数是对应的颜色，最后 2 个布尔型参数指定了这个颜色是否可写，以及当颜色不能分配时是否尽量找到一个“最匹配的”颜色。如果使用了最匹配的颜色而不是分配一个新颜色，颜色的 RGB 值会变成最匹配的值。可写颜色表的缺点较多，在应用时可能会产生兼容性的问题，因此应该尽量避免分配可写的颜色值。

如果 gdk_colormap_alloc_color 函数返回 TRUE，然后分配了一个颜色，则 color.pixel 中包含了一个有效的值，这个颜色就可以用于绘画了。

获得 RGB 值的另一种方法是使用 gdk_color_parse 函数，对其标准调用格式说明如下：

```
gint gdk_color_parse(gchar* spec,GdkColor* color)
```

这个函数采用 X 颜色规范，填充 GdkColor 的红、绿、蓝值。X 颜色规范可以有多种形式，其中一种可能形式是 RGB 字符串：

```
RGB:FF/FF/FF
```

其指定了一个白色（红绿蓝全部是全亮度）。“RGB:”用于指定一个“颜色空间”，并决定后面数字的意义。如果颜色规范字符串不是以一个可识别的“颜色空间”开头， X 系统假定它是一个颜色名，并在一个颜色名称数据库中查找。

当使用完一种颜色后，可以用 gdk_colormap_free_colors 函数将它从颜色表中删除，对其标准调用格式说明如下：

```
void gdk_colormap_free_colors(GdkColormap* colormap,GdkColor* colors,gint ncolors)
```

第 1 个参数是可绘区的颜色表，第 2 个参数是要删除的颜色，第 3 个参数是要删除的颜色个数。

获得颜色表的方法就是使用 gtk_widget_get_colormap 函数，对其标准调用格式说明如下，其参数是可绘图区：

```
GdkColormap* gtk_widget_get_colormap      (GtkWidget       *widget)
```


注 意

Linux 系统（缺省）的颜色表通常就是想要的，调用 gdk_colormap_get_system 函数时不需要参数，它返回回缺省的颜色表。

2. GTK+的事件处理

GTK+中的事件用来通知系统在什么时间应该进行绘图操作，事件传送到应用程序中以指明在一个 GdkWindow 中的变化或者有意义的用户动作。所有的事件都与一个 GdkWindow 相关联。它们也与一个 GtkWidget 相关联，GTK+主循环将事件从 GDK 传递给 GTK+构件树。

GTK+中的事件与之类似，它们被称为低级事件，GTK+中有信号与这些低级事件相联系。这些信号的处理函数有额外的参数，该函数是一个结构指针，包含事件的信息。例如，传递给设置事件（configure_event）处理函数的参数是一个 GdkEventConfigure 类型的结构指针，它的定义如下：

```
typedef struct _GdkEventConfigure      GdkEventConfigure;
struct _GdkEventConfigure
{
   GdkEventType type;
   GdkWindow *window;
   gint8 send_event;
   gint x, y;
   gint width;
   gint height;
};
```

type 会设置为事件的类型，如设置事件是 GDK_CONFIGURE，window 是发生事件的窗口。如果 send_event 为 TRUE，则表示事件由应用程序引发，如果为 FALSE，则由 X 服务器引发。在应用程序中可以通过声明一个静态的 event 结构来“虚构”一个事件，在结构中填充相应的值，然后引发与事件相对应的构件信号。这些合成的事件中 send_event 应当设置为 TRUE。x 和 y 用于给出事件的坐标，width 和 height 给出了需要设置的宽度和高度。

函数 g_signal_connect 用于来决定事件发生时调用的处理函数。在调用该函数之前，需要让 GTK+知道应用程序想处理的事件，可以使用函数 gtk_widget_set_events，对其标准调用格式说明如下，其中第 2 个参数是程序想处理的事件，其常用值如表 13.12 所示。

```
void gtk_widget_set_events (GtkWidget   *widget, gint     events);
```

表 13.12　events 的取值

events 值	对应事件类型
GDK_EXPOSURE_MASK	GDK_EXPOSE（曝光事件）
GDK_POINTER_MOTION_MASK	GDK_MOTION_NOTIFY（鼠标移动事件）
GDK_BUTTON_MOTION_MASK	GDK_MOTION_NOTIFY（鼠标键按下并且移动）
GDK_BUTTON1_MOTION_MASK	GDK_MOTION_NOTIFY（鼠标左键按下并且移动）
GDK_BUTTON2_MOTION_MASK	GDK_MOTION_NOTIFY（鼠标右键按下并且移动）
GDK_BUTTON3_MOTION_MASK	GDK_MOTION_NOTIFY（鼠标中间键按下并且移动）
GDK_BUTTON_PRESS_MASK	GDK_BUTTON_PRESS（鼠标按键被按下）
GDK_BUTTON_RELEASE_MASK	GDK_BUTTON_RELEASE（鼠标按键按下后按键被释放）
GDK_KEY_PRESS_MASK	GDK_KEY_PRESS（键盘按下）

（续表）

events 值	对应事件类型
GDK_KEY_RELEASE_MASK	GDK_KEY_RELEASE（键盘释放）
GDK_ENTER_NOTIFY_MASK	GDK_ENTER_NOTIFY（鼠标进入窗口）
GDK_LEAVE_NOTIFY_MASK	GDK_LEAVE_NOTIFY（鼠标离开窗口）
GDK_STRUCTURE_MASK	GDK_CONFIGURE（设置事件）
GDK_FOCUS_CHANGE_MASK	GDK_FOCUS_IN，GDK_FOCUS_OUT（窗口获得焦点，失去焦点）

13.2.3　俄罗斯方块的代码设计

俄罗斯方块游戏的应用代码分为后台处理和用户界面两大部分，前者用于对游戏中各个部件的安放、消除、翻转等操作进行处理，而后者用于绘制对应的图形。

对于后台处理部分而言，游戏的核心数据结构是一个 m*n 的矩阵。每种样式的方块出现时，都占据着矩阵中的几个位置。根据这些被占据的位置，可以把矩阵相应位置的值设置为 1。没有小方块占据的地方，矩阵的值就是 0。同样，落到底部的方块占据位置的矩阵元素值也都设置为 1。每到一定时间，定时器超时，或用户按下按键，上方的砖头状态就要发生变化，或左右移动，或翻转，或下降。这个动作能否成功，取决于目标位置有没有障碍物，也就是说方块的目标区域，有没有已经被设置为 1 的矩阵元素，或超出游戏区域边界。如果不冲突，根据方块的行为，调整矩阵的数值就可以了。

对于界面显示处理而言，界面的刷新是由定时器超时，或用户按下按键而触发的。每次相应事件发生的时候，对应的矩阵元素数值就会发生变化。界面刷新时不管是什么原因造成矩阵数值变化，它只是简单地根据现在矩阵的数值，把用户界面重新绘制一遍。更优化一些的算法可以是当定时器超时，或有用户输入按键时，这些事件把自己修改的矩阵范围传递给界面刷新程序。这样，界面刷新程序可以根据这个范围，只更新失效的界面区域，而不用把整个界面都重新更新。

根据实际设计需求可以将整个代码划分为数据结构，以及常数定义模块、后台处理模块、界面处理模块和菜单处理模块 4 个部分。

【例 13.4】俄罗斯方块的数据结构和常数定义模块

俄罗斯方块的数据结构和常数定义模块用于定义与程序相关的数据结构和常数，在其中定义了如下 4 个结构体。

- Position：表示方块中小方块的坐标。
- Block：表示 4 个小方块各自坐标以及方块所在矩形区域的起始位置，该矩形区域用来通知界面需要更新的区域。
- Brick：4 个小方块的坐标位置（以原点为参考点）以及方块所在矩形区域的起始位置，其中 index 分量用于表示方块的当前形态。
- KeyArg：保存程序主窗口，以及游戏绘图区和下一方块提示绘图区的指针。

实例的应用代码如下：

```
1   ifndef _GLOBAL_H
2   #define _GLOBAL_H
3   #include <gnome.h>
4   #include <gtk/gtk.h>
5   #define XMAX       10                    /*游戏区域 X 坐标最大值*/
6   #define YMAX       20                    /*游戏区域 Y 坐标最大值*/
7   #define BLOCKWIDTH 20                    /*方块中每一小方块的宽度*/
8   #define BLOCKHEIGHT 20                   /*方块中每一小方块的高度*/
9   /*游戏区域宽度和高度*/
10  #define GAMEAREAWIDTH ( XMAX*BLOCKWIDTH)
11  #define GAMEAREAHEIGHT ( YMAX*BLOCKHEIGHT)
12  /*下一方块提示区宽度和高度*/
13  #define NEXTAREAWIDTH     140
14  #define NEXTAREAHEIGHT    120
15  #define NUMBRICK      7                  /*方块总的类型数*/
16  /*分别消掉 1、2、3、4 行所得分数*/
17  #define ONEROWSCORE           1
18  #define TWOROWSCORE           3
19  #define THREEROWSCORE       7
20  #define FOURROWSCORE        13
21  typedef struct _Position {
22          gint x;
23          gint y;
24  } Position;
25  /*方块中 4 个小方块的各自坐标以及方块所在矩形区域的起始位置*/
26  typedef struct _block {
27          Position blockpos[4];
28          Position startpos;
29          Position endpos;
30  } Block;
31  /*每种方块的 4 种形态*/
32  typedef struct _brick{
33          Block brick[4];
34          gint index;
35  }Brick;
36  /*保存程序主窗口，以及绘图区和下一方块提示绘图区参数*/
37  typedef struct _KeyArg {
38          GtkWidget *window;
39          GtkWidget *game_area;
40          GtkWidget *nextbrick_area;
41  } KeyArg;
42  #endif
```

【例 13.5】俄罗斯方块的后台处理模块

俄罗斯方块的后台处理模块由一个.h 头文件和一个.c 源代码文件组成，前者用于声明后者中使用的几个函数，对这些函数及其功能说明如下。

- time_handler：定时器超时处理函数。

- PositionCorrect：判断方块是否到达边界以及目标位置是否已存在方块。
- KeyPress：实现按键处理。
- initBrick：初始化 7 种方块中的 4 种形态下的坐标，以及所在矩形区域的起始结束位置坐标。

这是后台处理模块的头文件，主要用于函数声明。

```
1  #ifndef     _CONTROL_H
2  #define     _CONTROL_H
3  #include "global.h"
4  gboolean time_handler(GtkWidget *widget);
5  gint PositionCorrect(gint newind,gint srcx, gint srcy);
6  gint KeyPress(GtkWidget *widget, GdkEventKey *event, gpointer arg);
7  void initBrick();
8  #endif
```

以下是后台处理模块的 C 语言源文件代码，首先是使用 g_Filled 对游戏区域网格的划分和声明，如果网格中填充有方块，则对应的网格值为 1，否则为 0；g_curbrickx 和 g_curbricky 变量用于标记方块在游戏区域的网格坐标；time_handler 函数用于实现定时器超时处理，在处理函数中虚拟一个向下的按键事件，然后将事件放入 GTK+的事件队列中。

```
1  #include "control.h"
2
3  extern guint tid;
4  extern guint mytimer;
5  extern GtkWidget *level_label,*score_label,*line_label;
6
7  gint g_Filled[XMAX][YMAX];              /*标记游戏区域是否有方块*/
8  gboolean bStop=TRUE;                    /*标识游戏是否结束*/
9  gboolean bPause=FALSE;                  /*标识游戏是否暂停*/
10 guint nLevel,nLine,nScore;              /*游戏等级、消去的行数以及所得总分数*/
11 Brick g_allbrick[ NUMBRICK];            /*所有方块及其默认形态*/
12 Brick g_curbrick;                       /*当前方块及其形态*/
13 Brick g_nextbrick;                      /*下一方块及其形态*/
14
15 gint g_curbrickind;                     /*当前方块形态索引*/
16 gint g_nextbrickind;                    /*下一方块形态索引*/
17 gint g_curbrickx=3;                     /*方块初始位置 x 坐标*/
18 gint g_curbricky=0;                     /*方块初始位置 y 坐标*/
19
20 /*定时器超时处理函数*/
21 gboolean time_handler(GtkWidget *widget)
22 {
23    GdkEvent t_event;
24
25    if (widget->window == NULL || TRUE==bStop)
26          return FALSE;
27      t_event.type=GDK_KEY_PRESS;
```

```
28         ((GdkEventKey*)&t_event)->window=widget->window;
29         ((GdkEventKey*)&t_event)->send_event=TRUE;
30         ((GdkEventKey*)&t_event)->time=0;
31         ((GdkEventKey*)&t_event)->state=0;
32         ((GdkEventKey*)&t_event)->keyval=GDK_Down;
33         ((GdkEventKey*)&t_event)->length=4;
34         ((GdkEventKey*)&t_event)->string="Down";
35         ((GdkEventKey*)&t_event)->hardware_keycode=0;
36         ((GdkEventKey*)&t_event)->group=0;
37         gdk_event_put(&t_event);
38         return TRUE;
39     }
```

PositionCorrect 函数用于实现方块目标区域的检测功能，其输入参数是方块形态索引值，以及目标区域的坐标，然后把方块当前所在区域网格清 0，再根据输入参数取得目标区域坐标值，并判断目标区域是否到达边界，以及目标区域是否已经存在方块，最后返回判断结果。

```
1     /*检测方块是否到达边界以及目标区域是否已有方块*/
2     gint PositionCorrect(gint newind,gint srcx, gint srcy)
3     {
4
5         gint i, sum=0,ind=g_curbrick.index;
6         gint x1,y1,x, y;
7
8         /*把方块当前区域清除*/
9         for(i=0;i<4;i++){
10            x=g_curbrick.brick[ind].blockpos[i].x+g_curbrickx;
11            y=g_curbrick.brick[ind].blockpos[i].y+g_curbricky;
12            g_Filled[x][y]=0;
13        }
14        /* 目标区域标记为当前方块，判断在目标区域是否已有方块以及是否到达边界*/
15        for(i=0;i<4;i++){
16            x1=g_curbrick.brick[newind].blockpos[i].x;
17            y1=g_curbrick.brick[newind].blockpos[i].y;
18            x1+=srcx;
19            y1+=srcy;
20            if(x1<0||x1>=XMAX||y1<0||y1>=YMAX)
21                sum++;
22            else
23                sum+=g_Filled[x1][y1];
24        }
25
26        /* 恢复方块到原来占用区域*/
27        for(i=0;i<4;i++){
28            x=g_curbrick.brick[ind].blockpos[i].x+g_curbrickx;
29            y=g_curbrick.brick[ind].blockpos[i].y+g_curbricky;
30            g_Filled[x][y]=1;
31        }
32        /* 返回判断结果 */
```

```
33        if(sum>=1)
34            return 0;    //碰到边界
35        else
36            return 1;
37    }
```

KeyPress 函数用于实行按键处理，其可以分为如下几个部分：

- 对回车键的处理：实现游戏暂停/继续之间的转换。
- 对空格键的处理：首先是把当前方块的索引值加 1，然后判断是否满足目标区域的要求，如果满足要求，则把方块的索引值更新，并更新方块中 4 个小方块相应的坐标值，否则不进行更新。
- 对方向键下的处理：如果方块没有到达底部，则更新方块的坐标值，如果到达底部，则要判断底部是否已经达到游戏允许的最大高度，如果达到最大高度，则弹出游戏结束对话框，并初始化游戏区域和下一方块提示区域。如果底部没有达到最大高度，则从底部向上依次判断每一行是否被方块填充满，如果填充满，则消去此行，并将该行上方部分复制到下一行。然后根据本次消去的总行数判断所得分数并累加；根据消去的累加行数判断是否应该晋级，如果晋级则缩短定时间隔。再取消原来的定时器，并设置新定时器为新的定时间隔，更新成绩显示。
- 实现方向键的左、右处理，其处理方法与空格键类似。

```
1 gint KeyPress(GtkWidget *widget, GdkEventKey *event, gpointer arg)
2 {
3      gint ind,oldind,x,y,ret, i;
4      Position oldstart,oldend,newstart,newend;
5      GdkEvent t_event;
6      KeyArg *arg1=arg;
7      oldstart.x=g_curbrickx;
8      oldstart.y=g_curbricky;
9      ind=g_curbrick.index;
10       oldind=g_curbrick.index;
11       oldend=g_curbrick.brick[ind].endpos;
12       oldend.x+=g_curbrickx;
13       oldend.y+=g_curbricky;
14       ret=0;
15
16     if(bPause==TRUE && event->keyval!=GDK_Return)
17       return    0;
18       switch(event->keyval)
19       {
20            case GDK_Return:
21                  if(bPause==FALSE)
22                     bPause=TRUE;
23                  else
24                     bPause=FALSE;
25                   break;
```

```
26
27          case GDK_space:
28          {
29              ind=(g_curbrick.index+1)%4;
30               ret=PositionCorrect(ind,g_curbrickx, g_curbricky);
31               if(ret==1){
32                   for(i=0;i<4;i++){
33                       x=g_curbrick.brick[oldind].blockpos[i].x+g_curbrickx;
34                       y=g_curbrick.brick[oldind].blockpos[i].y+g_curbricky;
35                       g_Filled[x][y]=0;
36                   }
37                   g_curbrick.index=ind;    //设为新的 index
38                   for(i=0;i<4;i++){
39                       x=g_curbrick.brick[ind].blockpos[i].x+g_curbrickx;
40                       y=g_curbrick.brick[ind].blockpos[i].y+g_curbricky;
41                       g_Filled[x][y]=1;
42                   }
43               }
44          }
45            break;
46
47            case GDK_Down:
48            {
49               ret=PositionCorrect(ind,g_curbrickx, g_curbricky+1);
50               if(ret==1){
51                   for(i=0;i<4;i++){
52                       x=g_curbrick.brick[ind].blockpos[i].x+g_curbrickx;
53                       y=g_curbrick.brick[ind].blockpos[i].y+g_curbricky;
54                       g_Filled[x][y]=0;
55                   }
56                    g_curbricky++;
57                  for(i=0;i<4;i++){
58                       x=g_curbrick.brick[ind].blockpos[i].x+g_curbrickx;
59                       y=g_curbrick.brick[ind].blockpos[i].y+g_curbricky;
60                       g_Filled[x][y]=1;
61                   }
62               }
63               else{    /*到达底部*/
64                   ret=1;
65                   if(0==g_curbricky)   //到达最顶部，游戏结束
66                   {
67                       bStop=TRUE;
68                       GtkWidget *dialog;
69                       dialog = gtk_message_dialog_new((GtkWindow *)widget,
70                                   GTK_DIALOG_DESTROY_WITH_PARENT,
71                                   GTK_MESSAGE_ERROR,
72                                   GTK_BUTTONS_OK,
73                                   "游戏结束！ ");
74                       gtk_window_set_title(GTK_WINDOW(dialog), "游戏结束");
```

```
75                  gtk_dialog_run(GTK_DIALOG(dialog));
76                  gtk_widget_destroy(dialog);
77                  gamearea_configure(arg1->game_area,NULL);
78                  nextbrickarea_configure(arg1->nextbrick_area,NULL);
79                  return   1;
80              }
81
82              /*消掉被方块填满的行，同时将上方区域下移一行*/
83              gint srcy=YMAX-1;
84              gint dsty=YMAX-1;
85              guint fullRow = 0;      //
86              for(y=YMAX-1;y>=0;y--){
87                  if(srcy!=dsty){                         /*需要下移*/
88                          for( x=0;x<XMAX;x++){           /*复制需要移动的区域*/
89                              g_Filled[x][dsty]=g_Filled[x][srcy];
90                          }
91                  }
92                  srcy--;
93                  /*统计被方块填充满的行*/
94                  gint sumrow=0;
95                  for( x=0;x<XMAX;x++){
96                      sumrow+=g_Filled[x][dsty];
97                  }
98                  if(sumrow<XMAX)     /*该行未填充满*/
99                      dsty--;
100                 else
101                     ++fullRow;
102             }
103             for(y=0;y<=dsty;y++){
104                 for(x=0;x<XMAX;x++)
105                     g_Filled[x][y]=0;
106             }
107             /*根据消去的行数判断所得分数*/
108             switch (fullRow)
109             {
110                 case 0:
111                     break;
112                 case 1:
113                     nScore += ONEROWSCORE;
114                     break;
115                 case 2:
116                     nScore += TWOROWSCORE;
117                 break;
118                 case 3:
119                     nScore += THREEROWSCORE;
120                     break;
121                 case 4:
122                     nScore += FOURROWSCORE;
123                     break;
```

```
124                     default:
125                         g_print("错误!\n");
126                         return -1;
127                 }
128                  nLine+=fullRow;    //更新行数
129                 if(fullRow!=0) {
130                 if(nLine<=20)
131                 {
132                     nLevel=1;
133                     mytimer=1000;
134                   }
135                 else if(nLine<=40)
136                 {
137                     nLevel=2;
138                     mytimer=800;
139                 }
140                 else if(nLine <=60)
141                 {
142                     nLevel=3;
143                     mytimer=600;
144                 }
145                  else if(nLine<=8)
146                 {
147                     nLevel=4;
148                     mytimer=400;
149                 }
150                  else
151                  {
152                      nLevel=5;
153                      mytimer=200;
154                  }
155                 if(tid)
156                             g_source_remove    (tid);
157             tid=g_timeout_add(mytimer, (GSourceFunc) time_handler, (GtkWidget *) arg1->window);
158                   time_handler((GtkWidget *)arg1->window);
159                 }
160                 gchar buffer[20];
161                 sprintf (buffer, "等  级：%d",nLevel);
162                 gtk_label_set_text(GTK_LABEL(level_label),buffer);
163                 sprintf (buffer, "行  数：%d",nLine);
164                 gtk_label_set_text(GTK_LABEL(line_label),buffer);
165                 sprintf (buffer, "分  数：%d",nScore);
166                 gtk_label_set_text(GTK_LABEL(score_label),buffer);
167
168                 g_curbrickind=g_nextbrickind;
169                 g_curbrick=g_allbrick[g_curbrickind];
170                 g_nextbrickind=g_random_int_range(0, NUMBRICK);
171                 g_nextbrick=g_allbrick[g_nextbrickind];
172                 ind=g_curbrick.index;
```

```
173                         g_curbrickx=3;
174                         g_curbricky=0;
175                         for(i=0;i<4;i++){
176                                 x=g_curbrick.brick[ind].blockpos[i].x+g_curbrickx;
177                                 y=g_curbrick.brick[ind].blockpos[i].y+g_curbricky;
178                                 g_Filled[x][y]=1;
179                         }
180                         oldstart.x=0;oldstart.y=0;
181                         oldend.x=XMAX-1;oldend.y=YMAX-1;
182                 }
183           }
184           break;
185
186         case GDK_Right:
187         {
188                 ret=PositionCorrect(ind,g_curbrickx+1, g_curbricky);
189                 if(ret==1){
190                         for(i=0;i<4;i++){
191                                 x=g_curbrick.brick[ind].blockpos[i].x+g_curbrickx;
192                                 y=g_curbrick.brick[ind].blockpos[i].y+g_curbricky;
193                                 g_Filled[x][y]=0;
194                         }
195                         g_curbrickx++;
196                         for(i=0;i<4;i++){
197                                 x=g_curbrick.brick[ind].blockpos[i].x+g_curbrickx;
198                                 y=g_curbrick.brick[ind].blockpos[i].y+g_curbricky;
199                                 g_Filled[x][y]=1;
200                         }
201                 }
202         }
203         break;
204
205         case GDK_Left:
206         {
207                 ret=PositionCorrect(ind,g_curbrickx-1, g_curbricky);
208                 if(ret==1){
209                         for(i=0;i<4;i++){
210                                 x=g_curbrick.brick[ind].blockpos[i].x+g_curbrickx;
211                                 y=g_curbrick.brick[ind].blockpos[i].y+g_curbricky;
212                                 g_Filled[x][y]=0;
213                         }
214                         g_curbrickx--;
215                         for(i=0;i<4;i++){
216                                 x=g_curbrick.brick[ind].blockpos[i].x+g_curbrickx;
217                                 y=g_curbrick.brick[ind].blockpos[i].y+g_curbricky;
218                                 g_Filled[x][y]=1;
219                         }
220                 }
221         }
```

```
222                 break;
223             }
224             if(ret==1){
225                 newstart.x=g_curbrickx;
226                 newstart.y=g_curbricky;
227                 ind=g_curbrick.index;
228                 newend=g_curbrick.brick[ind].endpos;
229                 newend.x=newend.x+newstart.x;
230                 newend.y=newend.y+newstart.y;
231                 if(newstart.x>oldstart.x)     newstart.x=oldstart.x;
232                 if(newstart.y>oldstart.y)     newstart.y=oldstart.y;
233                 if(newend.x<oldend.x)     newend.x=oldend.x;
234                 if(newend.y<oldend.y)     newend.y=oldend.y;
235
236                 /*重绘游戏区*/
237                 t_event.type=GDK_EXPOSE;
238                 ((GdkEventExpose*)&t_event)->window=((GtkWidget* )arg1->game_area)->window;
239                 ((GdkEventExpose*)&t_event)->send_event=TRUE;
240                 ((GdkEventExpose*)&t_event)->area.x=newstart.x*BLOCKWIDTH;
241                 ((GdkEventExpose*)&t_event)->area.y=newstart.y*BLOCKHEIGHT;
242                 ((GdkEventExpose*)&t_event)->area.width=(newend.x-newstart.x+1)*BLOCKWIDTH;
243                 ((GdkEventExpose*)&t_event)->area.height=(newend.y-newstart.y+1)*BLOCKHEIGHT;
244                 ((GdkEventExpose*)&t_event)->region=gdk_region_rectangle(&((GdkEventExpose*)&t_event)->area);
245                 ((GdkEventExpose*)&t_event)->count=0;
246                 gdk_event_put(&t_event);
247
248                 /*  重绘下一方块区  */
249                 t_event.type=GDK_EXPOSE;
250                 ((GdkEventExpose*)&t_event)->window=((GtkWidget* )arg1->nextbrick_area)->window;
251                 ((GdkEventExpose*)&t_event)->send_event=TRUE;
252                 ((GdkEventExpose*)&t_event)->area.x=0;
253                 ((GdkEventExpose*)&t_event)->area.y=0;
254                 ((GdkEventExpose*)&t_event)->area.width=NEXTAREAWIDTH;
255                 ((GdkEventExpose*)&t_event)->area.height=NEXTAREAHEIGHT;
256                 ((GdkEventExpose*)&t_event)->region=gdk_region_rectangle(&((GdkEventExpose*)&t_event)->area);
257                 ((GdkEventExpose*)&t_event)->count=0;
258                 gdk_event_put(&t_event);
259             }
260             return 0;
261     }
```

initBrick 是方块初始化函数，使用二维数组 par 保存了 7 种方块的 4 种形态下的 4 个小方块坐标，以及方块所在矩形区域的开始位置和结束位置，最后对 7 种方块每种形态下的坐标以及所占用的矩形区域进行设置。

```
1   void initBrick()
2   {
3       /*每种方块在 4 种形态下的 4 个小方块坐标，以及方块所在矩形区域的开始位置和结束位置*/
4       gint      par[NUMBRICK][48]={{0,1,1,1,2,1,1,0,0,0,2,1,\
```

```
5                                        0,0,0,1,0,2,1,1,0,0,1,2,\
6                                         0,0,1,0,2,0,1,1,0,0,2,1,\
7                                         1,0,1,1,1,2,0,1,0,0,1,2},\
8
9                                        {0,0,1,0,2,0,3,0,0,0,3,0,\
10                                        0,0,0,1,0,2,0,3,0,0,0,3,\
11                                        0,0,1,0,2,0,3,0,0,0,3,0,\
12                                        0,0,0,1,0,2,0,3,0,0,0,3},\
13
14                                       {0,1,1,1,2,1,2,0,0,0,2,1,\
15                                        0,0,0,1,0,2,1,2,0,0,1,2,\
16                                        0,0,1,0,2,0,0,1,0,0,2,1,\
17                                        0,0,1,0,1,1,1,2,0,0,1,2},\
18
19                                       {0,0,1,0,0,1,1,1,0,0,1,1,\
20                                        0,0,0,1,1,0,1,1,0,0,1,1,\
21                                        0,0,1,0,0,1,1,1,0,0,1,1,\
22                                        0,0,0,1,1,0,1,1,0,0,1,1},\
23
24                                       {0,0,0,1,1,1,1,2,0,0,1,2,\
25                                        1,0,2,0,0,1,1,1,0,0,2,1,\
26                                        0,0,0,1,1,1,1,2,0,0,1,2,\
27                                        1,0,2,0,0,1,1,1,0,0,2,1},
28
29                                        {0,0,0,1,1,1,2,1,0,0,2,1,\
30                                         1,0,0,0,0,1,0,2,0,0,1,2,\
31                                         0,0,1,0,2,0,2,1,0,0,2,1,\
32                                         1,0,1,1,1,2,0,2,0,0,1,2},\
33
34                                        {1,0,1,1,0,1,0,2,0,0,1,2,\
35                                         0,0,1,0,1,1,2,1,0,0,2,1,\
36                                         1,0,1,1,0,1,0,2,0,0,1,2,\
37                                         0,0,1,0,1,1,2,1,0,0,2,1}} ;
38         int i,j,k,l;
39         for(i=0;i<NUMBRICK;i++)
40         {
41                 g_allbrick[i].index=0;
42                 for(j=0;j<4;j++)
43                 {
44                         for(k=0;k<4;k++)
45                         {
46                                 g_allbrick[i].brick[j].blockpos[k].x= par[i][j*12+2*k];
47                                 g_allbrick[i].brick[j].blockpos[k].y= par[i][j*12+2*k+1];
48                         }
49                         g_allbrick[i].brick[j].startpos.x=par[i][j*12+8];
50                         g_allbrick[i].brick[j].startpos.y=par[i][j*12+9];
51                         g_allbrick[i].brick[j].endpos.x=par[i][j*12+10];
52                         g_allbrick[i].brick[j].endpos.y=par[i][j*12+11];
53                 }
```

```
54          }
55          /*设定当前方块及下一方块*/
56          g_curbrickind=g_random_int_range(0, NUMBRICK);
57          g_curbrick=g_allbrick[g_curbrickind];
58          g_nextbrickind=g_random_int_range(0, NUMBRICK);
59          g_nextbrick=g_allbrick[g_nextbrickind];
60      }
```

【例 13.6】俄罗斯方块的界面处理模块

俄罗斯方块的界面处理模块同样由一个.h 头文件和一个.c 源文件组成，前者用于对后者中的函数进行声明，对这些函数说明如下。

- gamearea_configure：对游戏的区域界面进行初始化操作。
- nextbrickarea_configure：对下一个方块区域进行提示初始化操作。
- amearea_expose：用于更新游戏区域的显示。
- nextbirckarea_expose：负责下一方块提示区域的显示更新。

以下是界面处理模块的头文件。

```
1   #ifndef _DISPLAY_H
2   #define _DISPLAY_H
3   #include     "global.h"
4   gint gamearea_configure (GtkWidget *widget, GdkEventConfigure *event);
5   gint nextbrickarea_configure (GtkWidget *widget, GdkEventConfigure *event);
6   gint    gamearea_expose (GtkWidget *widget, GdkEventExpose *event);
7   gint    nextbirckarea_expose (GtkWidget *widget, GdkEventExpose *event);
8   #endif
```

以下为界面处理模块的 C 语言源文件代码，其首先设置事件处理函数，根据宽度和高度产生一个相应大小的后端位图，然后产生 2 个 gc，其前端颜色分别为橙色和绿色，其中橙色用来填充小方块，而绿色则是给小方块四周加上边框。随后是用系统默认的 gc 在后端位图上画出游戏区域的矩形，最后把后端位图复制到屏幕窗口上。

```
1   #include      "display.h"
2
3    extern gboolean bStop;
4    extern gint g_Filled[XMAX][YMAX];
5
6   GdkPixmap *game_pixmap,*nextbrick_pixmap;     /*游戏绘图区以及提示区后端位图*/
7   GdkColor color;
8   GdkColormap *colormap;
9   GdkGC *gc,*gc1;
10
11  /* 创建适当大小的游戏区及提示区后端位图 */
12  gint gamearea_configure (GtkWidget *widget, GdkEventConfigure *event)
13  {
14              if (game_pixmap)
```

```
15                  gdk_pixmap_unref(game_pixmap);
16              game_pixmap = gdk_pixmap_new(widget->window, GAMEAREAWIDTH,
17                                                  GAMEAREAHEIGHT, -1);
18              gc =gdk_gc_new(game_pixmap);
19               if (gdk_color_parse("orange", &color))
20               {
21                     if (gdk_colormap_alloc_color(colormap, &color, FALSE, TRUE))
22                         gdk_gc_set_foreground (gc, &color );
23              }
24
25               gc1 =gdk_gc_new(game_pixmap);
26                if (gdk_color_parse("green", &color))
27                  {
28                      if (gdk_colormap_alloc_color(colormap, &color, FALSE, TRUE))
29                          gdk_gc_set_foreground (gc1, &color );
30                  }
31               gdk_draw_rectangle (game_pixmap,    widget->style->white_gc, TRUE,    0, 0,
32                                      GAMEAREAWIDTH, GAMEAREAHEIGHT);
33              gdk_draw_pixmap(widget->window, widget->style->white_gc, game_pixmap,0,0,0,0,
34                                              GAMEAREAWIDTH, GAMEAREAHEIGHT);
35              return TRUE;
36  }
```

这是下一方块提示区设置事件处理函数。其实现与 gamearea_configure 函数类似，只不过没有创建 gc 的过程。该区域绘制时使用的 gc 与绘制游戏区时使用的 gc 相同，所以不需要再创建。

```
 1  gint nextbrickarea_configure (GtkWidget *widget, GdkEventConfigure *event)
 2  {
 3          if (nextbrick_pixmap)
 4                  gdk_pixmap_unref(nextbrick_pixmap);
 5          nextbrick_pixmap = gdk_pixmap_new(widget->window, GAMEAREAWIDTH,
 6                                          GAMEAREAHEIGHT, -1);
 7          gdk_draw_rectangle (nextbrick_pixmap,    widget->style->white_gc, TRUE,    0, 0,
 8                                      NEXTAREAWIDTH, NEXTAREAHEIGHT);
 9          gdk_draw_pixmap(widget->window, widget->style->white_gc, nextbrick_pixmap, 0 ,0, 0, 0,
10                              NEXTAREAWIDTH, NEXTAREAHEIGHT);
11          return TRUE;
12  }
```

这是游戏区曝光事件处理函数。首先判断游戏是否结束，如果没结束，则首先取得需要更新区域的坐标，然后在更新区域内，如果有小方块，则首先绘制一个橙色矩形，其内部是填充的，再绘制一个绿色矩形，其内部不填充，其效果就是给橙色矩形四周加了一个绿色边框。如果更新区域内没有小方块，则绘制一个白色矩形。如果游戏结束，则把游戏区域恢复成白色区域，当绘制完成后，再将其从后端位图复制到屏幕窗口。

```
 1  gint    gamearea_expose (GtkWidget *widget, GdkEventExpose *event)
 2  {
 3
```

```
4        gint x,y;
5        gint srcx, srcy, destx, desty;
6        if(FALSE==bStop)
7        {
8            srcx=event->area.x/BLOCKWIDTH;
9            srcy=event->area.y/BLOCKHEIGHT;
10           destx=(event->area.x+event->area.width-1)/BLOCKWIDTH;
11           desty=(event->area.y+event->area.height-1)/BLOCKHEIGHT;
12           for(y=srcy;y<=desty;y++){
13               for(x=srcx;x<=destx;x++){
14                   if(g_Filled[x][y]==1){                /*方块占据区域*/
15                       gdk_draw_rectangle (game_pixmap,gc, TRUE,
16                                   x*BLOCKWIDTH,y*BLOCKHEIGHT,
17                                   BLOCKWIDTH, BLOCKHEIGHT);
18                       gdk_draw_rectangle (game_pixmap,gc1, FALSE,
19                                   x*BLOCKWIDTH,y*BLOCKHEIGHT,
20                                   BLOCKWIDTH, BLOCKHEIGHT);
21                   }
22                   else{
23                        gdk_draw_rectangle (game_pixmap,widget->style->white_gc, TRUE,
24                                   x*BLOCKWIDTH,y*BLOCKHEIGHT,
25                                   BLOCKWIDTH, BLOCKHEIGHT);
26                  }
27              }
28           }
29       }
30       else
31           gdk_draw_rectangle (game_pixmap,   widget->style->white_gc, TRUE,   0, 0,
32                       GAMEAREAWIDTH, GAMEAREAHEIGHT);
33      gdk_draw_pixmap(widget->window, widget->style->white_gc, game_pixmap,
34                   event->area.x, event->area.y, event->area.x, event->area.y,
35                   event->area.width, event->area.height);
36
37      return FALSE;
38  }
```

以上是下一方块提示区曝光事件处理函数，其实现与 nextbirckarea_expose 类似。

```
1   gint   nextbirckarea_expose (GtkWidget *widget, GdkEventExpose *event)
2   {
3          gint g_x=2,g_y=2,x1,y1,index;
4          extern Brick g_nextbrick;
5
6            if(FALSE==bStop) {
7                gdk_draw_rectangle (nextbrick_pixmap,   widget->style->white_gc, TRUE,   0, 0,
8                            NEXTAREAWIDTH, NEXTAREAHEIGHT);
9                for(index=0;index<4;index++)
10               {
11                   x1=(g_nextbrick.brick[0].blockpos[index].x+g_x)*BLOCKWIDTH;
12                   y1=(g_nextbrick.brick[0].blockpos[index].y+g_y)*BLOCKHEIGHT;
```

```
13                          gdk_draw_rectangle (nextbrick_pixmap,gc, TRUE, x1,y1,
14                                                      BLOCKWIDTH, BLOCKHEIGHT);
15                          gdk_draw_rectangle (nextbrick_pixmap,gc1, FALSE, x1,y1,
16                                                      BLOCKWIDTH, BLOCKHEIGHT);
17                      }
18                  }
19                  else
20                      gdk_draw_rectangle (nextbrick_pixmap,   widget->style->white_gc, TRUE,   0, 0,
21                                                      NEXTAREAWIDTH, NEXTAREAHEIGHT);
22              gdk_draw_pixmap(widget->window, widget->style->white_gc, nextbrick_pixmap,
23                                          event->area.x, event->area.y, event->area.x, event->area.y,
24                                          event->area.width, event->area.height);
25          return FALSE;
26  }
```

【例 13.7】俄罗斯方块菜单处理模块

俄罗斯方块的菜单处理模块同样由.h 头文件和.c 源文件组成，前者用于声明在后者中使用的函数，对这些函数说明如下。

- NewGame: “新建游戏”菜单项的处理函数。
- HelpContent: “帮助”菜单中的“内容”菜单项的处理函数，激活该菜单项后，程序弹出“帮助内容”对话框。
- CloseContent: 用于关闭该对话框。
- About: 是“帮助”菜单中的“关于”菜单项的处理函数，激活该菜单项后，将弹出“关于”对话框。

以下是菜单处理模块的头文件。

```
1  #ifndef       _MENU_H
2  #define       _MENU_H
3  #include "global.h"
4  gint NewGame(GtkWidget *widget, gpointer data);
5  gint     CloseContent(GtkWidget *widget,gpointer data);
6  gint     HelpContent(GtkWidget *widget, gpointer data);
7  gint About(GtkWidget *widget, gpointer data);
8  #endif
```

以下是菜单处理模块的 C 语言源文件，首先初始化一些参数，然后调用 initBrick 函数对方块进行初始化，最后初始化界面并且设置定时器后启动游戏。

```
1  #include "menu.h"
2
3  extern gboolean time_handler(GtkWidget *widget);
4  extern gboolean bStop;                    /*标识游戏是否结束*/
5  extern gboolean bPause;                   /*标识游戏是否暂停*/
6  extern guint nLevel,nLine,nScore;         /*游戏等级、消去带行数以及所得总分数*/
7  extern gint g_curbricky;
```

```
 8  extern GtkWidget *nextbrick_label, *record_label,*level_label,*score_label,*line_label;
 9  extern gint g_Filled[XMAX][YMAX];
10
11  guint tid=0;                                   /*定时器 ID*/
12  guint mytimer=1000;                            /*定时器初始周期间隔*/
13
14  /*“新建游戏”菜单项处理函数*/
15  gint NewGame(GtkWidget *widget, gpointer data)
16  {
17      KeyArg *arg=(KeyArg *)data;
18      int x,y;
19      gchar buffer[20];
20      bStop=FALSE;
21      mytimer=1000;
22      nLine=0;
23      nScore=0;
24      nLevel=1;
25      g_curbricky=0;
26      for (x=0;x<XMAX;x++)
27          for(y=0;y<YMAX;y++)
28          g_Filled[x][y]=0;
29      initBrick();
30      gamearea_configure(arg->game_area,NULL);
31      nextbrickarea_configure(arg->nextbrick_area,NULL);
32
33      sprintf (buffer, "等  级：1");
34      gtk_label_set_text(GTK_LABEL(level_label),buffer);
35      sprintf (buffer, "行  数：0");
36      gtk_label_set_text(GTK_LABEL(line_label),buffer);
37      sprintf (buffer, "分  数：0");
38      gtk_label_set_text(GTK_LABEL(score_label),buffer);
39      if(tid)
40              g_source_remove   (tid);
41      tid=g_timeout_add(mytimer, (GSourceFunc) time_handler, (GtkWidget *) arg->window);
42      time_handler((GtkWidget *)arg->window);
43      return 0;
44  }
```

CloseContent 函数用于销毁对话框并且继续游戏。

```
1  gint    CloseContent(GtkWidget *widget,gpointer data)
2  {
3      bPause=FALSE;
4      gtk_widget_destroy(data);
5      return    0;
6  }
```

HelpContent 函数用于在一个文本视图构件中显示帮助内容。

```
1  /*“内容”菜单项处理函数*/
```

```
2   gint     HelpContent(GtkWidget *widget, gpointer data)
3   {
4       GtkWidget *dialog;
5     GtkWidget *hbox;
6     GtkWidget *view;
7     GtkWidget *halign;
8     GtkWidget *ok;
9       GtkTextBuffer *buffer;
10      GtkTextIter iter;
11      char   text[1024]=" \n    方向键左：向左移动方块\n    方向键右：向右移动方块\n\    方向键下：
        使方块加速向下   \n    空格键(Space)：旋转方块\n    回车键(Enter)：暂停&继续游戏\n";
12
13      bPause=TRUE;
14      dialog=gtk_dialog_new();
15      gtk_window_set_title(GTK_WINDOW(dialog), "俄罗斯方块--帮助");
16      gtk_widget_set_size_request (GTK_WIDGET (dialog), 400,250 );
17      hbox=gtk_hbox_new(TRUE,10);
18      /*创建文本视图构件，显示帮助说明*/
19      view=gtk_text_view_new();
20      gtk_text_view_set_editable((GtkTextView   *)view,FALSE);    /*文本视图构件不可编辑*/
21      gtk_text_view_set_cursor_visible((GtkTextView   *)view,FALSE);
        /*文本视图构件不显示光标*/
22      buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(view));
23      gtk_text_buffer_get_iter_at_offset(buffer, &iter, 0);
24              gtk_text_buffer_insert(buffer, &iter, text, -1);
25
26      gtk_box_pack_start (GTK_BOX( hbox),view, TRUE, TRUE, 10);
27      gtk_box_pack_start (GTK_BOX (GTK_DIALOG (dialog)->vbox),hbox, TRUE, TRUE, 10);
28      ok=gtk_button_new_with_label("确定");
29      halign = gtk_alignment_new(0.5, 0.5, 0, 0);
30      gtk_container_add(GTK_CONTAINER(halign), ok);
31      gtk_box_pack_start (GTK_BOX (GTK_DIALOG (dialog)->vbox),halign, FALSE, FALSE, 10);
32
33      g_signal_connect(G_OBJECT(dialog), "delete_event", G_CALLBACK(CloseContent), G_OBJECT(dialog));
34      g_signal_connect(G_OBJECT(ok), "clicked", G_CALLBACK(CloseContent), (gpointer) dialog);
35
36      gtk_widget_show_all (dialog);
37      return    0;
38  }
```

About 函数用于显示游戏的关于信息。

```
1   gint About(GtkWidget *widget, gpointer data)
2   {
3       GtkWidget *dialog = gtk_about_dialog_new();
4     gtk_about_dialog_set_name(GTK_ABOUT_DIALOG(dialog), "俄罗斯方块");
5     gtk_about_dialog_set_version(GTK_ABOUT_DIALOG(dialog), "0.9");
6     gtk_about_dialog_set_copyright(GTK_ABOUT_DIALOG(dialog),
7                                         "(Copyright) 刘学勇 \n 2011.11");
8     gtk_about_dialog_set_comments(GTK_ABOUT_DIALOG(dialog),
```

```
9                  "基于 GDK 的俄罗斯方块游戏\n  本程序仅用于教学，请勿用于商业目的。");
10      gtk_dialog_run(GTK_DIALOG (dialog));
11      gtk_widget_destroy(dialog);
12      return    0;
13  }
```

【例 13.8】俄罗斯方块的代码综合

俄罗斯方块游戏的代码综合如例 13.13 所示，其对应的窗口布局如图 13.4 所示。

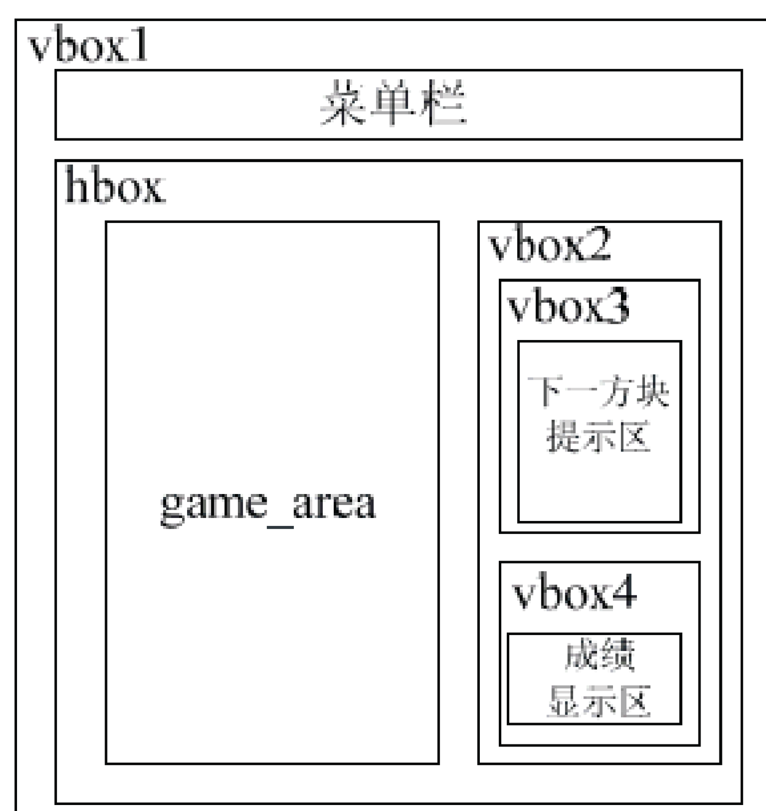


图 13.4　俄罗斯方块游戏的窗口布局

```
1   #i nclude "display.h"
2   #include "control.h"
3   #include "menu.h"
4
5   GtkWidget *game_area, *nextbrick_area;         /*游戏绘图区以及提示绘图区*/
6   GtkWidget *nextbrick_label, *record_label,*level_label,*score_label,*line_label;
7   KeyArg arg;
8
9   extern GdkColormap *colormap;
10
11  int main( int argc, char *argv[])
12  {
13      GtkWidget *window;
14     GtkWidget *vbox1,*vbox2,*vbox3,*vbox4,*hbox;
15
16      GtkWidget *menubar;
17      GtkWidget *gamemenu,    *helpmenu;
18      GtkWidget *game,*newgame, *sep, *quit;
19      GtkWidget *help,*content, *about;
20      GtkAccelGroup *accel_group = NULL;
21
22     gtk_init(&argc, &argv);
23
24    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
25    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
26    gtk_window_set_default_size(GTK_WINDOW(window), 500, 360);
27    gtk_window_set_title(GTK_WINDOW(window), "俄罗斯方块");
28
```

```
29      vbox1=gtk_vbox_new(FALSE,0);
30     vbox2=gtk_vbox_new(TRUE,0);
31     vbox3=gtk_vbox_new(FALSE,0);
32     vbox4=gtk_vbox_new(TRUE,0);
33     hbox=gtk_hbox_new(TRUE,20);
34
35      menubar = gtk_menu_bar_new();
36        /*建立“游戏”、“帮助”菜单项容器   */
37      gamemenu = gtk_menu_new();
38      helpmenu= gtk_menu_new();
39
40       /*建立加速键容器*/
41      accel_group = gtk_accel_group_new();
42      gtk_window_add_accel_group(GTK_WINDOW(window), accel_group);
43
44      /*“游戏”菜单中的各菜单项*/
45      game = gtk_menu_item_new_with_mnemonic("游戏(_G)");
46      newgame = gtk_menu_item_new_with_mnemonic("新建游戏(_N)");
47      sep = gtk_separator_menu_item_new();
48      quit = gtk_menu_item_new_with_mnemonic("退出(_Q)");
49
50       /*创建“游戏”菜单中的各菜单项加速键*/
51       gtk_widget_add_accelerator(newgame, "activate", accel_group,
52            GDK_N, GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
53      gtk_widget_add_accelerator(quit, "activate", accel_group,
54            GDK_Q, GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
55      /*“游戏”菜单中的各菜单项与“游戏”菜单关联*/
56      gtk_menu_item_set_submenu(GTK_MENU_ITEM(game), gamemenu);
57      gtk_menu_append(GTK_MENU_SHELL(gamemenu), newgame);
58      gtk_menu_append(GTK_MENU_SHELL(gamemenu), sep);
59      gtk_menu_append(GTK_MENU_SHELL(gamemenu), quit);
60
61        /*连接“游戏”菜单项处理函数*/
62        g_signal_connect (G_OBJECT(newgame),"activate",
63                                          G_CALLBACK(NewGame), &arg);
64        g_signal_connect(G_OBJECT(quit), "activate",
65                                          G_CALLBACK(gtk_main_quit), NULL);
66
67        /*创建“帮助”菜单中的各菜单项*/
68        help= gtk_menu_item_new_with_mnemonic("帮助(_H)");
69        content= gtk_menu_item_new_with_mnemonic("内容(_C)");
70        sep = gtk_separator_menu_item_new();
71        about= gtk_menu_item_new_with_mnemonic("关于(_A)");
72
73      /*“帮助”菜单中的各菜单项与“帮助”菜单关联*/
74      gtk_menu_item_set_submenu(GTK_MENU_ITEM(help), helpmenu);
75      gtk_menu_append(GTK_MENU_SHELL(helpmenu), content);
76      gtk_menu_append(GTK_MENU_SHELL(helpmenu),sep);
77      gtk_menu_append(GTK_MENU_SHELL(helpmenu), about);
```

```
78
79        g_signal_connect (G_OBJECT(content),"activate",
80                                              G_CALLBACK(HelpContent), NULL);
81       g_signal_connect (G_OBJECT(about),"activate",
82                                              G_CALLBACK(About), NULL);
83    /*把菜单放到菜单栏上*/
84     gtk_menu_bar_append(GTK_MENU_SHELL(menubar), game);
85     gtk_menu_bar_append(GTK_MENU_SHELL(menubar), help);
86        /*将菜单栏组装到 vbox1*/
87    gtk_box_pack_start(GTK_BOX(vbox1), menubar, FALSE, FALSE, 0);
88
89     /*创建游戏绘图区*/
90     game_area=gtk_drawing_area_new();
91     gtk_drawing_area_size ((GtkDrawingArea *)game_area,GAMEAREAWIDTH,GAMEAREAHEIGHT);
92     /*将绘图区组装到水平盒子*/
93     gtk_box_pack_start(GTK_BOX(hbox), game_area, FALSE, FALSE, 10);
94
95     /*下一方块提示区*/
96     nextbrick_label=gtk_label_new("下一方块");
97    gtk_box_pack_start(GTK_BOX(vbox3), nextbrick_label, FALSE, FALSE, 10);
98
99    /*创建显示下一方块绘图区*/
100   nextbrick_area=gtk_drawing_area_new();
101   gtk_drawing_area_size ((GtkDrawingArea *)nextbrick_area,NEXTAREAWIDTH,NEXTAREAHEIGHT);
102
103   gtk_box_pack_start(GTK_BOX(vbox3), nextbrick_area, FALSE, FALSE, 10);
104   gtk_box_pack_start(GTK_BOX(vbox2), vbox3, FALSE, FALSE, 0);
105
106  /*成绩区*/
107   record_label=gtk_label_new("成          绩");
108   level_label=gtk_label_new("等  级：");
109   line_label=gtk_label_new("行  数：");
110   score_label=gtk_label_new("分  数：");
111   gtk_box_pack_start(GTK_BOX(vbox4), record_label, FALSE, FALSE, 10);
112   gtk_box_pack_start(GTK_BOX(vbox4), level_label, FALSE, FALSE, 10);
113   gtk_box_pack_start(GTK_BOX(vbox4), line_label, FALSE, FALSE, 10);
114   gtk_box_pack_start(GTK_BOX(vbox4), score_label, FALSE, FALSE, 10);
115
116   gtk_box_pack_start(GTK_BOX(vbox2), vbox4, FALSE, FALSE, 10);
117   gtk_box_pack_start(GTK_BOX(hbox), vbox2, FALSE, FALSE, 10);
118   gtk_box_pack_start(GTK_BOX(vbox1), hbox, FALSE, FALSE, 20);
119   gtk_container_add(GTK_CONTAINER(window), vbox1);
120
121   colormap=gtk_widget_get_colormap(game_area);
122
123   gtk_widget_set_events (game_area, GDK_STRUCTURE_MASK|
124                                  GDK_EXPOSURE_MASK
125                                  |GDK_KEY_PRESS_MASK);
126
```

```
127    g_signal_connect (G_OBJECT(game_area),"configure_event",
128                                            G_CALLBACK(gamearea_configure), NULL);
129    g_signal_connect (G_OBJECT (game_area), "expose_event",
130                                             G_CALLBACK( gamearea_expose), NULL);
131     g_signal_connect (G_OBJECT(nextbrick_area),"configure_event",
132                                            G_CALLBACK(nextbrickarea_configure), NULL);
133    g_signal_connect (G_OBJECT (nextbrick_area), "expose_event",
134                                              G_CALLBACK( nextbirckarea_expose), NULL);
135          arg.window=window;
136          arg.game_area=game_area;
137          arg.nextbrick_area=nextbrick_area;
138         g_signal_connect(G_OBJECT(window),"key-press-event",
139                                            G_CALLBACK(KeyPress),(void*)&arg);
140
141       g_signal_connect_swapped(G_OBJECT(window), "delete_event",
142              G_CALLBACK(gtk_main_quit), G_OBJECT(window));
143
144     gtk_widget_show_all(window);
145     gtk_main();
146     return 0;
147  }
```

13.2.4 俄罗斯方块游戏的运行

俄罗斯方块游戏的启动界面如图 13.5 所示。

启动界面上有游戏和帮助两个顶层菜单，其中“游戏”包括“新建游戏”和“退出”两个菜单项，如图 13.6 所示。



图 13.5　俄罗斯方块游戏的启动界面



图 13.6　“游戏”菜单

单击“新建游戏”后将开始进行游戏，此时运行界面如图 13.7 所示。

另外一个顶级菜单“帮助”由“内容”和“关于”两个菜单项组成，如图 13.8 所示。



图 13.7　游戏的运行界面



图 13.8　“帮助”菜单

选择“内容”菜单项后，将弹出如图 13.9 所示的对话框。

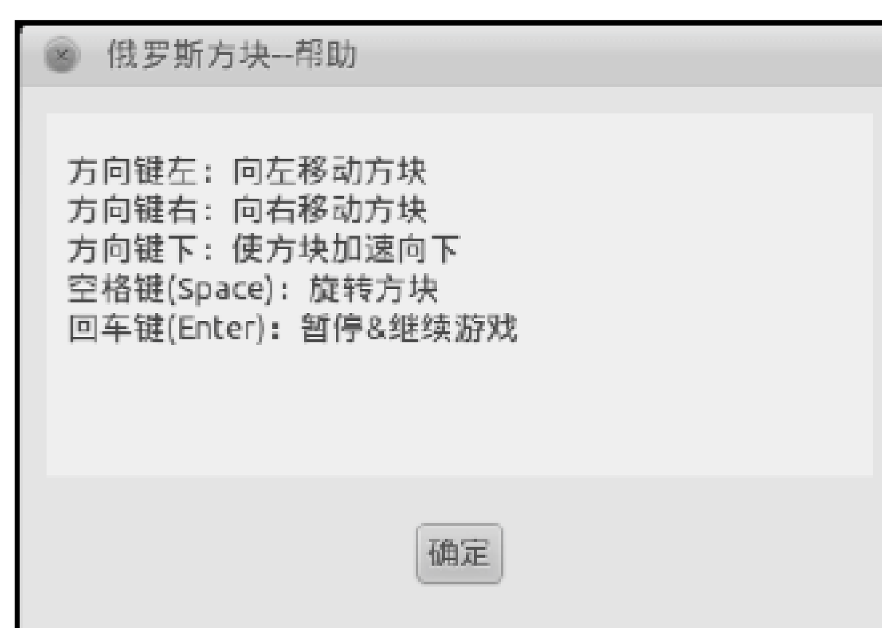


图 13.9　“俄罗斯方块-帮助”对话框

附录　习题答案

第 2 章

7. sqrtf 是平方根函数，对其标准调用格式说明如下（可以使用 man 命令获得更多的信息）：

```
#include <math.h>
float sqrtf(float x);
```

使用该函数以及 scanf 和 printf 函数实现从键盘输入 n 个实型数据，分别求其对应的平方根并且在屏幕上输出。

答案：

```
//键盘输入整数 n，然后输出 n 个实型数，求 n 个实型数的平方根
#include <stdio.h>
#include <math.h>
int main(void)
{
        int n,i;
        float x,y;
        scanf("%d",&n);                    //等待输入
        for(i=0;i<n;i++)                   //循环
        {
                scanf("%f",&x);            //输入 N 个数据
                y = sqrtf(x);              //求平方根
                printf("%f *****%f\n",x,y); //打印输出
        }
    return 0;
}
```

8. 使用 malloc 函数编写一段程序，用于模拟在内存中为一个手机的通讯录增加存储空间的情况，该通信录的结构体定义为 struct co，对其中各个分量说明如下。

- index：编号。
- name：姓名。
- MTel：手机号码。
- Tel：座机号码。

答案：

```
//在内存中添加一个单元
#include <stdio.h>
```

```
#include <stdlib.h>
#include <ctype.h>
struct co
{
    int index;
    char name[8];
    char MTel[12];
    char Tel[12];
};
int x;

int main(void)
{
    struct co *p;
    char ch;
    printf("do you add a user? Y/N\n");
    ch =getchar();
    if(ch == 'y'||ch == 'Y')
    {
            p = (struct co *)malloc(sizeof(struct co));
            p->index = ++x;
            printf("User name:");
            scanf("%s",p->name);
            printf("Mobile:");
            scanf("%s",p->MTel);
            printf("Home Tel:");
            printf("intex:%d\n name:%s\n MoveTel:%s\n HomeTel:%s\n",p->index,p->name,p->MTel,p->Tel);
    }
}
```

第 3 章

1. 使用 open 函数编写一个程序，打开或者创建一个指定的文件，带路径文件名由用户指定输入，文件名的最长长度为 30。

答案：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define FLAGS O_WRONLY | O_CREATE | O_TRUNC
#define MODE S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
int main(void)
{
     const char *pathname;                /*指向需要打开（或创建）文件的绝对路径名或相对路径名*/
```

```
        int fd;                                       /*文件描述符*/
        char pn[30];                                  /*pn 为字符串数组，存放要打开（或创建）的文件名*/
        printf("Input the pathname[<30 strings]:");   /*输入路径名，小于 30 个字符*/
        gets(pn);
        pathname=pn;
        if((fd=open(pathname,FLAGS,MODE))==-1)        /*调用 open 函数*/
          {
              printf("error,open file failed!\n");
              exit(1);                                /*出错退出*/
          }
        printf("OK,open file successful!\n");
        printf("fd=%d\n",fd);
        return 0;
    }
```

2. 编写一个程序，用于测试标准输入文件（文件描述符 0）是否能使用 lseek 函数来设置位移量。

答案：

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{
    if (lseek (0, 0, SEEK_CUR) == -1)      /*判断标准输入文件能否设置位移量*/
        printf("cannot seek!\n");
    else
        printf("seek OK!\n");
    exit(0);
}
```

3. 使用 write 函数来编写一个程序，在程序中指定一个文件，用户可以向程序中一次写入不超过 80 个字符的数据。

答案：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define FILENAME "/home/zhangfan/hello"            /*要进行写操作的文件*/
#define SIZE 80                                    /*定义缓冲区大小*/
#define FLAGS O_RDWR | O_APPEND
/*定义参数 flags：以读写方式打开文件，向文件添加内容时从文件尾开始写*/
```

```
int main(void)
{
    int count;
    int fd;                                    /*文件描述符*/
    char write_buf[SIZE];                      /*写缓冲区*/
    const char *pathname=FILENAME;             /*指向需要打开文件的路径名*/
    if((fd=open(pathname,FLAGS))==-1)          /*调用 open 函数打开文件*/
      {
          printf("error,open file failed!\n");
          exit(1);    /*出错退出*/
      }
    printf("OK,open file successful!\n");
    printf("Begin write:\n");
    gets(write_buf);
    count = strlen(write_buf);                 /*要写入文件的数据的字节数*/
    if (write(fd, write_buf, count)==-1)
{
         printf("error,write file failed!\n");
         exit(1);    /*写出错，退出*/
}
    printf("OK,write %d strings to file!\n",count);
 return 0;
}
```

4. 使用 read 和 write 函数，编写一个程序，实现 cp 函数的基础功能。

答案：

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#define PERMS 0666
#define    DUMMY 0
#define MAXSIZE 1024
int main(int argc, char *argv[])
{
   int sourcefileID, targetfileID;             //目标文件和源文件的描述符
   int readNO = 0;                             //读出的字符数
   char WRBuf[MAXSIZE];                        //定义缓冲区
   if(argc!=3)                                 //如果命令行参数不正确
   {
     printf("run error\n");
     return 1;
   }
   if((sourcefileID=open(*(argv+1),O_RDONLY,DUMMY))==-1)    //如果源文件打开失败
   {
     printf("Source file open error!\n");
     return 2;
   }
```

```
    if((targetfileID=open(*(argv+2), O_WRONLY|O_CREATE, PERMS))==-1)  //如果目标文件打开失败
    {
      printf("Target file open error!\n");
      return 3;
    }
    while((readNO=read(sourcefileID, WRBuf, MAXSIZE))>0)        //如果读出来的数据大于 0
    if(write(targetfileID, WRBuf,readNO)!=readNO)               //如果写入的数据和读出的数据不同
    {
      printf("Target file write   error!\n");                   //写数据错误
      return 4;
    }
    close(sourcefileID);
    close(targetfileID);   //关闭文件
    return 0;
}
```

5. 当使用 lseek 函数定位到超出文件尾端之后，对于新写入的数据需要分配磁盘块，但是对于原文件尾端和新开始写位置之间的部分则不需要分配磁盘块，这会产生空洞文件，文件中的空洞并不要求在磁盘上占有存储区，具体处理方式与文件系统的实现有关，请尝试使用 open、lseek 等函数创建一个含有空洞的文件。

答案：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define FILENAME "/home/zhangfan/test"             /*要进行操作的文件*/
#define FLAGS O_WRONLY | O_CREATE | O_TRUNC
/*定义参数 flags：以读写方式打开文件，向文件添加内容时从文件尾开始写*/
#define MODE 0600                                  /*定义参数 MODE ：文件所有者读写方式*/

int main(void)
{
char buf1[ ]={"abcdefghij"};                       /*缓冲区 1，长度为 10*/
    char buf2[ ]={"1234567890"};                   /*缓冲区 2，长度为 10*/
int fd;                                            /*文件描述符*/
int count;
    const char *pathname=FILENAME;                 /*指向需要进行操作的文件路径名*/
    if((fd=open(pathname,FLAGS,MODE))==-1)         /*调用 open 函数打开文件*/
     {
        printf("error,open file failed!\n");
        exit(1);    /*打开文件出错，退出*/
     }
count = strlen(buf1);                              /*缓冲区 1 的长度*/
```

```
        if(write(fd,buf1,count)!=count)                /*调用 write 函数将缓冲区 1 的数据写入文件*/
    {
            printf("error,write file failed!\n");
            exit(1);     /*写出错，退出*/
        }
        if(lseek(fd,50,SEEK_SET)==-1)
/*调用 lseek 函数定位文件，偏移量为 50，从文件开头计算偏移值*/
        {
            printf("error,lseek failed!\n");
            exit(1);     /*定位出错，退出*/
            }
        count = strlen(buf2);                          /*缓冲区 2 的长度*/
        if(write(fd,buf2,count)!=count)                /*调用 write 函数将缓冲区 2 的数据写入文件*/
            {
            printf("error,write file failed!\n");
            exit(1);     /*写出错，退出*/
            }
        return 0;
}
```

第 4 章

1．编写一个程序，将当前工作目录修改为用户指定的目录，然后调用 getcwd 命令获取并且打印当前的目录路径。

答案：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

#define SIZE 30                                /*定义缓冲区大小*/

int main(void)
{
char newpath[SIZE];
char buf[SIZE];
            printf("Input the new pathname[<30 strings]:");
    gets(newpath);
    if(chdir(newpath) == -1)                   /*调用 chdir 函数改变当前工作目录*/
      {
            printf("error,change directory failed!\n");
            exit(1);                           /*出错退出*/
            }
    printf("OK,change directory successful!\n");
        if(getcwd(buf,SIZE)==NULL)             /*调用 getcwd 函数获取当前工作目录*/
```

```
        {
            printf("error,getcwd failed!\n");
        exit(1);                                    /*出错退出*/
        }
printf("cwd = %s\n",buf);
        return 0;
}
```

2. 编写一个应用程序，首先用 getcwd 函数取得当前工作目录，然后在当前工作目录下，利用 mkdir 函数创建新目录。新目录创建成功后，改变当前工作目录为新目录，然后切换回上一级目录后删除新创建的目录。

答案：

```
    #include <sys/types.h>
    #include <unistd.h>
#include <limits.h>
    #include <sys/stat.h>

    int main(int argc, char *argv[])
    {
        char path[1000];
        char file[1000];
        if(argc!=2)
        {
            printf("Usage ex3-8 <pathname>\n");
            return 1;
        }
        getcwd(path);                                           /*取得当前工作目录 */
        printf("current directory is :%s \n",path);
        if(mkdir(argv[1],S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH)<0)    /*创建新目录 */
        {
            printf("mkdir failed \n");
            return 2;
        }
        if(chdir(argv[1])<0)                                /*改变当前工作目录为新目录 */
        {
            printf("chdir failed \n");
            return 3;
        }
        getcwd(path);
        printf("mkdir successed.\n New current directory is: %s\n",path);
       chdir(..);                                           //返回上一级目录
        rmdir(path);                                        /*删除新建目录 */
        printf("%s is removed\n",path);
        return 0;
    }
```

3. 编写一个程序，首先用 opendir 函数打开用户指定的目录，然后调用 readdir 函数读取该目

录内容并打印出所读目录内容，最后用 closedir 函数将刚才打开的目录文件关闭。

答案：

```
#include <sys/types.h>
#include <dirent.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
        char path[1000];
        DIR * dp;
        struct dirent *pdirent;
        if(argc!=2)
        {
                printf("Usage ex3-9 <pathname>\n");
                return 1;
        }
        if((dp=opendir(argv[1]))==NULL)
        {
                printf("Opendir %s failed\n", argv[1]);
                return 2;
        }
        if((pdirent=readdir(dp))==0)
        {
                printf("readdir %s failed\n", argv[1]);
                return 3;
        }
        printf("%s\n",pdirent->d_name);
  closedir(dp);
        return 0;
}
```

4. 编写一个程序，在某指定路径下创建一个空目录 temp，该目录文件的访问权限为用户可读可写可执行，同组用户可读可执行，其他组用户可读可执行，并返回函数执行的结果。

答案：

```
#include <sys/types.h>
#include <sys/stat.h>
#define PATHNAME "/home/alloy "                    //指定的路径
#define MODE S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH
int main(void)
{
    if(mkdir(PATHNAME, MODE)==-1)
    {
        printf("make dir error!\n");
        exit(1);
    }
    else
```

```
        printf("make dir successfuly!\n");
    return 0;
}
```

第 5 章

1. 编写一个程序，使用 stat 命令来获得/etc/passwd 文件的类型并且在屏幕上输出其大小。

答案：

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc,char *argv[])
{
   struct stat buf;
   stat("/etc/passwd", &buf);
   printf("/etc/passwd file size = %d \n", buf.st_size);
   return 0;
}
```

2. 编写一个程序，打开一个指定文件，将它截断至 0 长度，但维持它的访问时间和修改时间不变。

答案：

```
#include <sys/types.h>
#include <utime.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
        int i,fd;
        struct stat statbuf;
        struct utimbuf times;
        if(argc!=2)
        {
                printf("Usage: ex3-3 filename\n");
                return 1;
        }
        if((fd=open(argv[1],O_RDWR))<0)                /*打开文件      */
        {
                printf("%s open failed.\n",argv[1]);
                return 3;
        }
if(stat(argv[1],&statbuf)<0)
                return 2;

        if(ftruncate(fd,0)<0)                          /*截断文件      */
```

```
        {
            printf("%s truncate failed.\n",argv[1]);
            return 4;
        }
        printf("%s is truncated now.\n",argv[1]);
        times.actime=statbuf.st_atime;                    /*恢复文件时间至原时间*/
        times.modtime=statbuf.st_mtime;
        if(utime(argv[1], &times)==0)
            printf("utime() call successful \n");
        else
            printf("Error:utime() call failed. \n");
        return 0;
    }
```

3. 编写一个程序，用于测试 umask 函数的返回值。

答案：

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc,char *argv[])
{
  int a,b;
  a = umask(0);
  printf("%o\n",a);
  b = umask(a);
  printf("%o\n",b);
  printf("%o\n",umask(0026));
  printf("%o\n",umask(0066));
  return 0;
}
```

4. 编写一个程序，用于测试如果 rename 函数的 oldname 和 newname 参数使用同一个参数的时候 rename 的返回值。

答案：

```
#include <stdio.h>

int main(int argc,char **argv)
{
  if(argc < 3)
  {
   printf("Usage: %s old_name new_name\n",argv[0]);
   return -1;
  }
  printf("%s => %s\n", argv[1], argv[2]);
  if(rename(argv[1], argv[2]) < 0 )
```

```
    printf("error!\n");
    else
    printf("ok!\n");
    return 0;
}
```

5. 编写一个程序，使用 dup 函数复制一个已经用 open 函数打开的文件描述符，然后将其关闭。

答案：

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
int main(int argc, char * argv[])
{
        int fileID;
        if(argc!=2)
        {
                printf("arg error\n");  //参数错误
                return 1;
        }
        if((fileID = open(argv[1],O_WRONLY|O_CREATE,0644))==-1)
        {
                printf("open %s error\n",argv[1]);  //打开函数错误
                return 2;
        }
        printf("the old fileID is %d\n",fileID);
        if(fileID = dup(fileID)==-1)            //获得新的文件描述符
        {
                printf("dup error\n");          //dup 函数操作错误
                return 3;
        }
        printf("dup call successed!\n");        //dup 操作成功
        printf("the new fileID is %d\n",fileID); //打印新的文件描述符
        close(fileID);
        return 0;
 }
```

第 6 章

1. 编写一个程序，从键盘输入一个字符，并将其显示出来，当输入 q 时，程序退出。

答案：

```
#include <stdio.h>
int main(int argc, char *argv[])
{
```

```
    char c;
    while((c = getchar()) != 'q')
    {
        putchar(c);
    }
  return 0;
}
```

2. 编写一个程序，查看本机中标准输入流的缓冲区类型。

答案：

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("stdin is ");
    if(stdin->_flags & _IO_UNBUFFERED)          /*判断标准输入流对象的缓冲区类型*/
        printf("unbuffered\n");                 /*无缓存*/
    else if(stdin->_flags & _IO_LINE_BUF)
        printf("line-buffered\n");              /*行缓存*/
    else
        printf("fully-buffered\n");             /*全缓存*/
    printf("buffer size is %d\n", stdin->_IO_buf_end - stdin->_IO_buf_base);
                                                /*打印缓冲区的大小*/
    printf("file discriptor is %d\n\n", fileno(stdin));   /*标准输入流的文件描述符*/
    return 0;
}
```

3. 编写一个程序，从键盘中输入字符，并将它写入一个文件，当输入 q 时，程序退出。

答案：

```
#include <stdio.h>
int main(int argc, char *argv[] )
{
char ch;
FILE* fo = fopen("temp.txt", "wt");
while ( scanf("%c", &ch) != EOF && ch != 'q' )
fprintf(fo, "%c", ch);
fclose( fo );
return 0;
}
```

4. 编写一个程序，利用 sprintf 函数把二进制数据转换为十进制字符串的形式。

答案：

```
    #include <stdio.h>
    #include <math.h>
    int main(int argc, char* argv[])
    {
```

```
        char buf[80];
        float radius, area;
        float pi=3.1419926;
        if(argc!=2)
        {
            printf("Usage: %s radius \n",argv[0]);
            return 1;
        }
        sscanf(argv[1],"%f",&radius);          /*从命令行读入半径 */
        area=pi*radius*radius;
        sprintf(buf,"The area of a circle with radius %f is %f\n", radius, area);
        puts(buf);
        return 0;
    }
```

5. Linux 中的 wc 命令的功能为统计指定文件中的字节数、字数、行数，并将统计结果显示输出，编写一个程序，对标准输入设备（键盘）实现 wc 命令的功能。

答案：

```
    #include <stdio.h>
    #define BEGIN 1;                    /*开始读一个新单词，置为 BEGIN */
    int main(int argc, char *argv[])
    {
    int c, characters, lines, words, state;
    /*这些变量分别保存 getchar 函数的返回值、字符数、行数、单词数，
    记录是不是开始分析一个新单词的状态 */
    /*设置初始值 */
    state=0;
    characters=words=lines=0;
    /* 每次标准输入设备读入一个字符，直到输入字符‘0’ */
        while((c=getchar())!='0')
        {
            characters++;                 /* 单词数加一 */
            switch(c)
            {
            case '\n':
                lines++;                  /* 行数加一 */
                state=0;                  /* 新行标志，表示单词的结束 */
                break;
            case ' ':
                state=0;                  /* 空格字符表示单词的结束*/
                break;
            case '\t':
                state=0;                  /* 制表符表示单词的结束 */
                break;
            default:                      /* 其他情况，在某个单词中 */
                if(state==0)
                    {
```

```
                    state=BEGIN;
                    words++;        /* 如果是旧单词的结束，开始一个新单词*/
                }
            break;
        }
    }
    printf("There is %d characters, %d words, %d lines. \n",characters, words, lines);
    *输出结果 */
}
```

6. 在某文件中将学生的各种信息都存放在一个如下的结构体中：

```
struct {
char name[NAMESIZE];
long number;
short department;
short scores[10];
   } student;
```

编写一个程序，将存放学生各种信息的文件中的学生信息读出，重新组成一个存放所有学生的前 3 门成绩的文件。

答案：

```
#include <stdio.h>

#define NAMESIZE 30

struct {
char name[NAMESIZE];
long number;
short department;
short scores[10];                /*保存学生成绩的数组 */
} student;                       /*保存一个学生信息的结构 */

short *pscores;                  /*保存学生成绩的数组 */

int main(int argc, char *argv[])
{
    FILE *fpstudents;            /*已经存在的学生信息文件 */
    FILE *fpscore;               /*未存在的学生信息文件 */
    /*判断命令行输入是否正确 */
    if(argc<=2)
    {
        printf("usage: %s sourcefile destfile\n",argv[0]);
        return 1;
    }
    /*打开学生信息文件，判断是否出错。*/
    if((fpstudents=fopen(argv[1],"r"))==NULL)
    {
```

```
            printf("Open    sourcefile %s failed!", argv[1]);
            return 2;
        }
        /*创建学生成绩文件，判断是否出错。*/
        if((fpscore=fopen(argv[2],"w"))==NULL)
        {
            printf("Create    destfile %s failed!", argv[2]);
            return 3;
        }
        /*将成绩的前 3 项写入文件中  */
        while(fread(&student,sizeof(student),1,fpstudents)==1)
        {
            pscores=student.scores;
            if(fwrite(&pscores,sizeof(short),3,fpscore)!=3)
                printf("Error in writing file.\n");
            return 4;
        }
        return 0;
    }
```

第 7 章

1. 编写一个程序，使用 getpid 函数来获取当前进程的进程 ID。

答案：

```
#include <sys/types.h>
#include <stdio.h>
int main(int argc,char *argv[])
{
   printf("The current process ID is %d\n",getpid());
   return 0;
}
```

2. 编写一个程序，使用 fork 函数来创建一个子进程，并分别输出父、子进程的进程 ID。

答案：

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main (int argc,char *argv[])
{
    pid_t pid;                                  /*此时仅有一个进程*/
    if((pid=fork()) < 0)
      {
          printf("error in fork!");
          exit(1);                              /* fork 出错退出*/
      }
```

```
        else if(pid==0)
            printf("Child process ID is %d\n", getpid());
        else
            printf("Parent process ID is %d\n", getpid());
        return 0;
}
```

3. 编写一个程序，使用 fork 函数来创建一个子进程，并且说明父进程和子进程的随机返回问题。

答案：

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc,char *argv[])
{
    pid_t pid;
    printf("Start of fork testing. \n");
    pid=fork();
    printf("Return of fork success:pid=%d\n",pid);
    return 0;
}
```

4. 编写一个程序，使用 vfork 函数来创建一个子进程，并且说明父进程和子进程的随机返回问题。

答案略。

5. 编写一个程序，考察 exit 函数的使用方法，在程序尚未运行到最后时使用 exit 函数退出，查看后面的程序语句是否会被执行？

答案：

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc,char *argv[])
{
        printf("Hello, world!\n");
        exit(0);
        printf("Hello, world again!!\n");
}
```

6. 编写一个程序，考察_exit 函数的使用方法，在程序运行到最后时使用_exit 函数退出，查看程序的执行结果会发生怎样的变化。

答案略。

7. 编写一个程序，使用 getpgrp 函数来获取当前进程的进程组 ID 并且输出。

答案：

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc,char *argv[])
{
   printf("The current process ID is %d\n",getpgrp());
   return 0;
}
```

第 8 章

1. 编写一个程序，使用 signal 函数忽略从终端键入“Ctrl+\”时产生的 SIGQUIT 信号。

答案：

```
#include <signal.h>
#include <stdio.h>
int main(int argc,char *argv[])
{
      signal(SIGQUIT,SIG_IGN);
      while(1)
           sleep(1);
      return 0;
}
```

2. 编写一个程序，使用 raise 函数向进程自身发送一个 SIGABRT 信号，使进程非正常结束。

答案：

```
#include <sys/types.h>
#include <signal.h>
int main(int argc,char *argv[])

{
     printf("This is a test!\n");
     if(raise(SIGABRT) == -1)
     {
        printf("Send Failed!\n");
        exit(1);
     }
     printf("This is a test,again!\n");
     return 0;
}
```

3. 编写一个程序，使用 pause 函数将进程挂起，直到有 SIGALRM 信号发生时才从 pause 返回。

答案：

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void SignHandler(int iSignNo)
{
    printf("signal:%d\n",iSignNo);
}

int main(int argc,char *argv[])
{
    signal(SIGALRM,SignHandler);
    alarm(5);
    printf("Before pause().\n");
    pause();
    printf("After pause().\n");
    return 0;
}
```

4. 编写一个程序，使用信号，读入终端输入的字符，并将其中的小写字母转换成大写字母后输出。

答案：

```
#include <stdio.h>
#include <signal.h>

void sigcatcher(int signum);

int main(int argc,char *argv[])
{
    char buffer1[100], buffer2[100];
    int i;
    if(signal(SIGTERM, &sigcatcher)==-1)
    {
        printf("Couldn't register signal hanlder!\n");
        exit(1);
    }
    printf("Pid of This Process : %d \n",getpid());
    printf("Please input:\n");
    for(;;)
    {
        fgets(buffer1, sizeof(buffer1),stdin);
        for(i=0;i<100;i++)
        {
            if(buffer1[i]>=97&&buffer1[i]<=122)
                buffer2[i]=buffer1[i]-32;
            else
```

```
                    buffer2[i]=buffer1[i];
                }
                printf("Your input is: %s \n",buffer2);
        }
        exit(0);
}

void sigcatcher(int signum)
{
        printf("catch signal SIGTERM.\n");
        exit(0);
}
```

5. 编写一个程序，实现同一个信号处理函数对多个信号的处理。

答案：

```
#include <stdio.h>
#include <signal.h>

void intfunc(int signum);
void exitfunc(int signum);

int main(int argc,char *argv[])
{
        char buffer1[100],buffer2[100];
        int i;
        if(signal(SIGINT, &intfunc)==-1)
        {
                printf("Couldn't register signal hanlder for SIGINT!\n");
                exit(1);
        }
        if(signal(SIGTSTP, &intfunc)==-1)
        {
                printf("Couldn't register signal hanlder for SIGTSTP!\n");
                exit(1);
        }
        if(signal(SIGTERM, &exitfunc)==-1)
        {
                printf("Couldn't register signal hanlder for SIGTERM!\n");
                exit(1);
        }
        printf("Pid of This Process : %d \n",getpid());

        for(;;)
        {
                printf("Please input:\n");
                fgets(buffer1, sizeof(buffer1),stdin);
                for(i=0;i<100;i++)
                {
```

```
                    if(buffer1[i]>=97&&buffer1[i]<=122)
                        buffer2[i]=buffer1[i]-32;
                    else
                        buffer2[i]=buffer1[i];
                }
                printf("Your input is: %s \n",buffer2);
        }
        exit(0);
}

void intfunc(int signum)
{
        printf("catch signal %d \n",signum);
}

void exitfunc(int signum)
{
        printf("signal SIGTERM \n");
        exit(0);
}
```

6. 编写一个程序，为进程打印 SIGINT 和 SIGTERM 信号的掩码。

答案：

```
#include <stdio.h>
#include <signal.h>
#include <string.h>

 void pr_mask(int signum)
{
        sigset_t sigset;
        if(sigprocmask(0, NULL, &sigset)<0)
        {
                printf("sigprocmask error");
                exit(0);
        }
        if(sigismember(&sigset, SIGINT))
                printf("SIGINT \n");
        if(sigismember(&sigset, SIGTERM))
        {
               printf("SIGTERM \n");
               exit(0);
        }
}

int main(int argc,char *argv[])
{
        char buffer1[100],buffer2[100];
        int i;
```

```
        if(signal(SIGINT, &pr_mask)==-1)
        {
            printf("Couldn't register signal hanlder for SIGINT!\n");
            exit(1);
        }

        if(signal(SIGTERM, &pr_mask)==-1)
        {
            printf("Couldn't register signal hanlder for SIGTERM!\n");
            exit(1);
        }
        printf("Pid of This Process : %d \n",getpid());

        for(;;)
        {
            printf("Please input:\n");
            fgets(buffer1, sizeof(buffer1),stdin);
            for(i=0;i<100;i++)
            {
                if(buffer1[i]>=97&&buffer1[i]<=122)
                    buffer2[i]=buffer1[i]-32;
                else
                    buffer2[i]=buffer1[i];
            }
            printf("Your input is: %s \n",buffer2);
        }
        exit(0);
    }
```

第 9 章

1. 假设存在一个可以从标准输入读入字母并且将其从小写转换为大写输出的可执行程序 upcase，使用 pipe 函数编写一个应用程序，实现将一个指定文本文件中的字母都转换为大写。

答案：

```
    #include <unistd.h>
    #include <stdio.h>
    #include <sys/types.h>
    #include <sys/wait.h>

    int main(int argc, char *argv[])
    {
        int n,fd[2];
        pid_t pid;
        char buffer[BUFSIZ+1];
        FILE *fp;
```

```
if(argc<=1)
{
        printf("usage: %s <pathname>\n",argv[0]);
        exit(1);
}
/* 打开文本文件 */
if((fp=fopen(argv[1],"r"))==NULL)
{
        printf("Can't open %s \n", argv[1]);
        exit(1);
}
/* 创建管道 */
if(pipe(fd)<0)
{
        printf("pipe failed!\n ");
        exit(1);
}
/* 创建子进程 */
if((pid=fork())<0)
{
        printf("fork failed!\n ");
        exit(1);
}
else if (pid>0)                         /* 父进程 */
{
        close(fd[0]);
        while(fgets(buffer,BUFSIZ,fp)!=NULL)
        {
                n=strlen(buffer);
                /* 向管道中写入数据 */
                if(write(fd[1],buffer,n)!=n)
                {
                        printf("write error to pipe.\n");
                        exit(1);
                }
        }
        if(ferror(fp))
        {
                printf("fgets error. \n");
                exit(1);
        }
        close(fd[1]);
        if(waitpid(pid, NULL, 0)<0)
        {
                printf("waitpid error!\n");
                exit(1);
        }
        exit(0);
}
```

```
        else                                    /* 子进程 */
        {
            close(fd[1]);
            if(fd[0]!=STDIN_FILENO)
            {
                                                /* 将管道复制到标准输入 */
            if(dup2(fd[0],STDIN_FILENO)!=STDIN_FILENO)
                {
                    printf("dup2 error to stdin! \n");
                    exit(1);
                }
                close(fd[0]);
            }
            /* 运行用户程序 */
            if(execl("upcase","upcase",(char *)0)<0)
            {
                printf("execl error for upcase.\n");
                exit(1);
            }
            exit(0);
        }
    }
```

2. 使用 popen 函数来重写上一题的应用程序。

答案略。

3. 编写一个程序，使用 pipe 函数创建一个匿名管道，并使用 write 向管道的一端写入数据，使用 read 函数从管道的另一端读取数据。

答案：

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <limits.h>
#define BUFSIZE PIPE_BUF
int main(void)
{
    int fd[2];
    char buf[BUFSIZE] = "hello, Linux world!\n";
    pid_t pid;
    int len;
    if((pipe(fd)) < 0)                          /*创建管道*/
    {
        printf("pipe failed\n");
        exit(1);
    }
    if((pid = fork()) < 0)                      /*创建一个子进程*/
    {
        printf("fork failed\n");
```

```
        exit(1);
    }
    else if(pid > 0)
    {
        close ( fd[0] );                        /*父进程中关闭管道的读出端*/
        write (fd[1], buf, strlen(buf));        /*父进程向管道写入数据*/
        exit (0);
    }
    else
    {
        close ( fd[1] );                        /*子进程关闭管道的写入端*/
        len = read (fd[0], buf, BUFSIZE);       /*子进程从管道中读出数据*/
        if (len < 0)
        {
            printf("process failed when read a pipe\n");
            exit(1);
        }
        else
            write(STDOUT_FILENO, buf, len);     /*输出到标准输出*/
        exit(0);
    }
}
```

4. 编写一个程序，使用 msgget 函数创建一个消息队列，并返回该消息队列的描述符。

答案：

```
#include <sys/types.h>
#include <sys/ipc.h>
int main(void)
{
int gflags;
key_t key;
int msgid;
char *msgpath="/unix/msgqueue";
gflags=IPC_CREATE|IPC_EXCL;
key=ftok(msgpath,'a');                          /*获取消息队列键值*/
if((msgid=msgget(key,gflags|00666))==-1)
{
        printf("msg create error\n");
        return;
}
printf("msgid is %d\n",msgid);
return 0;
}
```

5. 编写一个程序，使用 msgsnd 函数向消息队列中发送一个字符串数据信息“Hello!This is a test!”，并通过查看消息队列的属性信息检验发送是否成功。

答案：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
int main(void)
{
int gflags,sflags,rflags;
key_t key;
int msgid;
int reval;
struct msgsbuf
{
        int mtype;
        char mtext[20];
    }msg_sbuf;                              /*发送消息缓冲区的数据结构*/
struct msqid_ds msg_ginfo,msg_sinfo;
char* msgpath="/unix/msgqueue";
key=ftok(msgpath,'a');                      /*获取消息队列的键值*/
gflags=IPC_CREATE|IPC_EXCL;
msgid=msgget(key,gflags|0666);              /*调用 msgget 创建消息队列*/
if(msgid==-1)
{
         printf("msg create error\n");
         return;
}
sflags=IPC_NOWAIT;                          /*消息队列满时，msgsnd 不等待，立刻出错返回*/
msg_sbuf.mtype=10;
msg_sbuf.mtext[20]= " Hello！ This is a test!";      /*将要发送的消息数据*/
reval=msgsnd(msgid,&msg_sbuf,sizeof(msg_sbuf.mtext),sflags); /*调用 msgsnd 发送消息*/
if(reval==-1)
{
         printf("message send error\n");
}
rerutn 0;
}
```

6. 编写一个程序，使用 semget 函数创建一个信号量集，并返回该信号量集的描述符。

答案：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SEM_PATH "/unix/my_sem"
int main(void)
{
    int flag1,key;
    flag=IPC_CREATE|IPC_EXCL|00666;
```

```
        if((key=ftok(SEM_PATH,'a')) == -1)
        {
            perror("ftok error");
            exit(1);
        }
        if((semid=semget(key,1,flag))<0)
        {
            perror("semget");
            exit(1);
        }
        printf("semid is %d\n",semid);
        return 0;
}
```

7. 编写一个程序，使用 write 函数向共享内存中写入数据，实现不同进程间的数据信息传递。

答案：

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>

typedef struct
{
          char name[4];
          int age;
} people;

int main(int argc, char** argv)
{
int shm_id,i;
          char temp;
          people *p_map;
if ( argc != 2 )                                            /*命令行参数错误*/
            {
            printf ("USAGE: atshm <identifier>" );          /*打印帮助消息*/
            exit (1);
      }
    shm_id = atoi(argv[1]);                                 /*得到要引入的共享内存段*/
          p_map=(people *)shmat(shm_id,NULL,0);
temp='a';
          for(i = 0;i<10;i++)
            {
              temp+=1;
              memcpy((*(p_map+i)).name,&temp,1);
              (*(p_map+i)).age=20+i;
            }
          if(shmdt(p_map)==-1)
    {
```

```
            perror("detach error!\n");
        }
    return 0;
}
```

第 10 章

1. 编写一个程序，调用 pthread_create 函数创建两个线程，一个打印自己的线程 ID 和“Hello”，另一个打印自己的线程 ID 号和“Ubuntu！”。

答案：

```
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void print_message(char*ptr);

int main()
{
        pthread_t thread1, thread2;
        char *msg1="Hello\n";
        char *msg2="Ubuntu!\n";
        pthread_create(&thread1,NULL, (void *)(&print_message), (void *)msg1);
        pthread_create(&thread2,NULL, (void *)(&print_message), (void *)msg2);
        sleep(1);
        return 0;
}

void print_message(char *ptr)
{
  int retval;
  printf("Thread ID: %lx", pthread_slef());
        printf("%s",ptr);
  pthread_exit(&retval);
}
```

2. 使用 pthread_join 函数来重写上一题的程序，使得两个线程重复打印 10 次，主进程等待两个线程都打印完成之后才退出。

答案：

```
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void print_msg(char *ptr);
```

```
int main()
{
        pthread_t thread1, thread2;
        int i,j;
        void *retval;
        char *msg1="Hello\n";
        char *msg2="World\n";
        pthread_create(&thread1,NULL, (void *)(&print_msg), (void *)msg1);
        pthread_create(&thread2,NULL, (void *)(&print_msg), (void *)msg2);
        pthread_join(thread1,&retval);
        pthread_join(thread2,&retval);
        return 0;
}

void   print_msg(char *ptr)
{
        int i;
        for(i=0;i<10;i++)
        printf("%s",ptr);
}
```

3．编写一个程序，使用 pthread_create 函数循环创建 5 个线程，然后每次在创建线程时将当前循环计数器的值通过 pthread_create 函数的 arg 参数传递给新线程，在线程中打印输出该计数器的值。

答案：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
//线程处理函数
void *threaddeal(void *arg)
{
   printf("%d\n",*((int *)arg));    //传递线程的参数
   pthread_exit(NULL);
}

int main(int argc,char *argv[])
{
 int i;
 pthread_t threadid;
 for(i=0;i<10;i++)
 {
 if(pthread_create(&threadid,NULL,threaddeal,&i) != 0)    //将 i 值作为参数传递
 {
    //若返回值不为 0，则表明创建线程失败
    printf("创建线程失败.\n");
    exit(0);       //退出
```

```
        }
      }
      pthread_exit(NULL);
      return 0;
    }
```

4. 编写一个程序，创建 0~4 共 5 个线程，然后每个线程输出一个 hello。

答案：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define    NUM_THREADS    8

void *PrintHello(void *args)
{
    int thread_arg;
    sleep(1);
    thread_arg = (int)args;
    printf("Hello from thread %d\n", thread_arg);
    return NULL;
}

int main(void)
{
    int rc,t;
    pthread_t thread[NUM_THREADS];

    for( t = 0; t < NUM_THREADS; t++)
    {
        printf("Createing thread %d\n", t);
        rc = pthread_create(&thread[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR; return code is %d\n", rc);
            return EXIT_FAILURE;
        }
    }
    for( t = 0; t < NUM_THREADS; t++)
        pthread_join(thread[t], NULL);
    return EXIT_SUCCESS;
}
```

5. 编写一个程序，实现一个线程从共享的缓冲区中读数据，另一个线程向共享的缓冲区中写数据，对共享的缓冲区的访问控制是通过使用一个互斥锁来实现的。

答案：

```
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

#define FALSE 0
#define TRUE 1

void readfun();
void writefun();

char buffer[256];
int buffer_has_item=0;
int retflag=FALSE;
pthread_mutex_t mutex;

int main()
{
        pthread_t reader;
        pthread_mutex_init(&mutex,NULL);
        pthread_create(&reader,NULL,(void *)&readfun,NULL);
        writefun();
}

void readfun()
{
        while(1)
        {
                if(retflag)
                        return;
                pthread_mutex_lock(&mutex);
                if(buffer_has_item==1)
                {
                        printf("%s",buffer);
                        buffer_has_item=0;
                }
                pthread_mutex_unlock(&mutex);
        }
}
void writefun()
{
        int i=0;
        while(1)
        {
                if(i==10)
                {
                        retflag=TRUE;
```

```
                        return;
                    }
                    pthread_mutex_lock(&mutex);
                    if(buffer_has_item==0)
                    {
                        sprintf(buffer,"This is %d\n",i++);
                        buffer_has_item=1;
                    }
                pthread_mutex_unlock(&mutex);
                }
        }
```

第 11 章

1. 编写一个程序，输出当前系统的信息（包括 CPU、操作系统和版本）以及使用的网络字节顺序。

答案：

```
#include <sys/utsname.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc,char *argv[])
{
    union
    {
        short inum;
        char c[sizeof(short)];
    } un;
    struct utsname            uts;
    un.inum = 0x0102;
    if(uname(&uts)<0)
    {
        printf("Could not get host information .\n");
        return -1;
    }
    printf("%s -%s-%s:\n",uts.machine, uts.sysname, uts.release);
    if(sizeof(short)!=2)
    {
        printf("sizeof short =%d\n", sizeof(short));
        return 0;
    }
    if(un.c[0]==1 && un.c[1]==2)
        printf("big_endian.\n");
    else if(un.c[0]==2 && un.c[1]==1)
        printf("little_endian.\n");
    else
```

```
        printf("unknown .\n");
    return 0;
}
```

2. 编写一个程序，获取本机的 IP 地址并且将其转换为网络地址。

参考第 11.2 节的例 11.2~11.6。

3. 编写一个程序，使用 socket()函数创建一个 TCP 套接口，并返回该套接字的描述符。

答案：

```
#include<sys/types.h>
#include<sys/socket.h>
int main(int argc,char *argv[])
{
    int sockfd;                                             /*定义套接口描述符*/
    if((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)          /*建立一个 socket*/
    {
        printf ("socket created error!");
        exit(1);
    }
    else                                                    /*socket 创建成功*/
        printf("socket id:%d\n",sockfd);
    return 0;
}
```

4. 编写一个程序，使用 socket()函数创建一个 UDP 套接口，并返回该套接字的描述符。

答案：

```
#include<sys/socket.h>
int main(void)
{
    int sockfd;                                             /*定义套接口描述符*/
    if((sockfd = socket(AF_INET,SOCK_DGRAM,0))<0)           /*建立一个 socket*/
    {
        printf ("socket created error!");
        exit(1);
    }
    else                                                    /*socket 创建成功*/
        printf("socket id:%d\n",sockfd);
    return 0;
}
```

5. 编写一对服务器-客户端的应用程序，客户端使用 TCP 向服务器端请求日期和时间，服务器端在收到请求后，回答请求并显示出客户的地址。

答案：

服务器端：

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define LISTENQ       5
#define       MAXLINE       512

int main()
{
        int listenfd, connfd;
        socklen_t   len;
        struct sockaddr_in servaddr, cliaddr;
        char  buff[MAXLINE];
        time_t        ticks;
        listenfd=socket(AF_INET, SOCK_STREAM,0);
        if(listenfd<0)
        {
                printf("Socket created failed.\n");
                return -1;
        }
        servaddr.sin_family=AF_INET;
        servaddr.sin_port=htons(6666);
        servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
        if(bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr))<0)
        {
                printf("bind failed.\n");
                return -1;
        }
        printf("listening....\n");
        listen(listenfd, LISTENQ);
        while(1)
        {
                len=sizeof(cliaddr);
                connfd=accept(listenfd,(struct sockaddr *)&cliaddr, &len);
                printf("connect from %s, port %d \n",inet_ntoa(cliaddr.sin_addr.s_addr),ntohs(cliaddr.sin_port));
                ticks=time(NULL);
                sprintf(buff,"% .24s \r \n",ctime(&ticks));
                write(connfd,buff,strlen(buff));
                close(connfd);
        }
}
```

客户端：

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <netdb.h>
#define     MAXBUFFSIZE256
#define     PORT 6666
#define HOST_ADDR "127.0.0.1"
int main(int argc, char *argv[])
{
        int sockfd,n;
        char recvbuff[MAXBUFFSIZE];
        struct sockaddr_in      servaddr;
        sockfd=socket(AF_INET,SOCK_STREAM,0);
        if(sockfd<0)
        {
                printf("Socket created failed.\n");
                return -1;
        }

        servaddr.sin_family=AF_INET;
        servaddr.sin_port=htons(6666);
        servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
        printf("connecting...\n");
        if(connect(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr))<0)
        {
                printf("Connect server failed.\n");
                return -1;
        }
        while((n=read(sockfd,recvbuff,MAXBUFFSIZE))>0)
        {
                recvbuff[n]=0;
                fputs(recvbuff,stdout);
        }
        if(n<0)
        {
                printf("Read failed!\n");
                return -2;
        }
        return 0;
}
```

6. 使用 UDP 协议实现上一题的功能。

答案略。

第 12 章

1. 编写一个程序，创建一个最简单的 GTK+窗口，并为窗口设置标题。

参考例 12.1。

2. 编写一个程序，创建如下图所示的窗口，其中文本框中可输入的最多字符数为 20，当单击“提交”按钮时在终端输出对应的输入字符串。



答案：

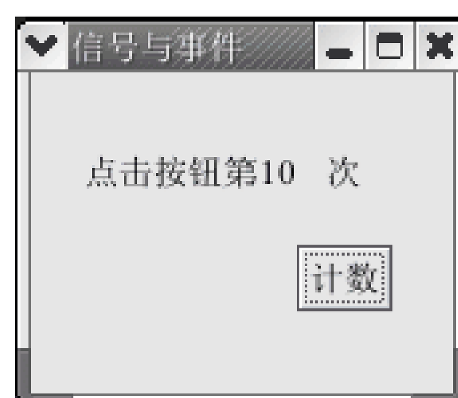
```
#include <gtk/gtk.h>
#include <stdlib.h>

GtkWidget *window;
GtkWidget *table;
GtkWidget *entry;   /*定义一个指向文本框的指针*/
GtkWidget *label;
GtkWidget *button;
char text[50];

void on_clicked(GtkWidget *widget, gpointer data)   /*定义回调函数*/
{
    strcpy(text, gtk_entry_get_text (GTK_ENTRY (entry)));
      /*获取文本框的文本内容，并将其复制到 text 字符串数组中*/
    printf("您输入的字符串是: %s\n",text);   /*打印文本框中输入的字符串*/
}

int main (int argc,char *argv[])
{
    gtk_init(&argc,&argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window),
    g_locale_to_utf8("文本框的使用",-1,NULL,NULL,NULL));
    /*为窗口设置标题，g_locale_to_utf8()函数支持中文字符显示*/
    table=gtk_table_new (3,2,FALSE);
    /*定义一个 3 行 2 列的表格，单元格大小会根据单元格中的元件大小自动调整*/
    label=gtk_label_new (g_locale_to_utf8("请在这里输入文本:",-1,NULL,NULL,NULL));
    entry = gtk_entry_new_with_max_length (20);
    /*新建一个有字数限制的文本框，文本框中最多可输入 20 个字符*/
    button=gtk_button_new_with_label (g_locale_to_utf8("提交",-1,NULL,NULL,NULL));
    /*新建“提交”按钮*/
    gtk_container_add (GTK_CONTAINER (window), table); /*将表格添加到窗口中*/
    /*下面是将三个元件分别添加到表格的相应位置中*/
    gtk_table_attach (GTK_TABLE(table), label, 0, 1, 0, 1,
                    (GtkAttachOptions)(0), (GtkAttachOptions)(0), 10, 10);
    gtk_table_attach (GTK_TABLE(table), entry, 0, 2, 1, 2,
                    (GtkAttachOptions)(0), (GtkAttachOptions)(0), 10, 10);
    gtk_table_attach (GTK_TABLE(table), button, 1, 2, 2, 3,
                    (GtkAttachOptions)(0), (GtkAttachOptions)(0), 10, 10);
    g_signal_connect (G_OBJECT(button), "clicked", G_CALLBACK(on_clicked), window);
    /*为“提交”按钮添加信号回调函数*/
    gtk_widget_show_all (window);      /*显示窗口中的所有元件*/
    gtk_main();
    return 0;
}
```

3. 编写一个程序，创建如下窗口，单击“计数”按钮的时候显示当前单击按钮的次数。



答案：

```
#include <gtk/gtk.h>
#include <stdlib.h>

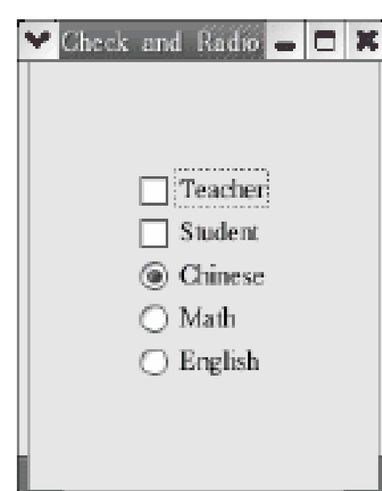
int i=0;
GtkWidget *window;                                  /*指向窗口的指针*/
GtkWidget *table;                                   /*指向表格的指针*/
GtkWidget *label1;                                  /*指向文本框的指针*/
GtkWidget *label2;                                  /*指向文本框的指针*/
GtkWidget *label3;                                  /*指向文本框的指针*/
GtkWidget *button;                                  /*指向按钮的指针*/

void on_clicked(GtkWidget *widget, gpointer data)   /*定义信号回调函数*/
{
    char a[20];
    i++;
    gcvt ((float)i,3,a);
    gtk_label_set_text (GTK_LABEL (label2), a);     /*设置按钮的标签*/
}

int main (int argc,char *argv[])
{
    gtk_init(&argc,&argv);
    window=gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window),
                        g_locale_to_utf8("信号与事件",-1,NULL,NULL,NULL));
    gtk_container_border_width (GTK_CONTAINER(window),20);
    table=gtk_table_new(2,3,FALSE);
    /*定义一个 2 行 3 列的表格，单元格大小会根据单元格中的元件大小自动调整*/
    label1=gtk_label_new (g_locale_to_utf8("点击按钮第",-1,NULL,NULL,NULL));
    label2=gtk_label_new (g_locale_to_utf8("0",-1,NULL,NULL,NULL));
    label3=gtk_label_new (g_locale_to_utf8("次",-1,NULL,NULL,NULL));
    button = gtk_button_new_with_label(g_locale_to_utf8("计数",-1,NULL,NULL,NULL));
    /* “计数”按钮*/
    gtk_container_add (GTK_CONTAINER (window), table); /*将表格添加到窗口中*/
    gtk_table_attach (GTK_TABLE(table), label1, 0, 1, 0, 1,
                    (GtkAttachOptions)(0), (GtkAttachOptions)(0), 0, 10);
    gtk_table_attach (GTK_TABLE(table), label2, 1, 2, 0, 1,
                    (GtkAttachOptions)(0), (GtkAttachOptions)(0), 0, 10);
    gtk_table_attach (GTK_TABLE(table), label3, 2, 3, 0, 1,
                    (GtkAttachOptions)(0), (GtkAttachOptions)(0), 0, 10);
    gtk_table_attach (GTK_TABLE(table), button, 2, 3, 1, 2,
                    (GtkAttachOptions)(0), (GtkAttachOptions)(0), 0, 10);
    g_signal_connect (G_OBJECT(button), "clicked", G_CALLBACK(on_clicked), window);
    gtk_widget_show_all (window); /*显示窗口*/
```

```
    gtk_main();
    return 0;
}
```

4. 编写一个程序，创建一个如下图所示的窗口，包括 3 个单选按钮和 2 个复选框，3 个单选按钮的选项分别为 Chinese、Math 和 English，2 个复选框的选项为 Teacher 和 Student。



答案：

```
#include<gtk/gtk.h>
int main (int argc,char *argv[])
{
GtkWidget *window;
GtkWidget *box;
GSList *group;
GtkWidget *check,*radio;
char title[]="Check and Radio";
gtk_init (&argc,&argv);
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_window_set_title (GTK_WINDOW (window), title);
gtk_container_border_width (GTK_CONTAINER(window),50);
box = gtk_vbox_new (FALSE,0);
gtk_container_add(GTK_CONTAINER(window),box);
check = gtk_check_button_new_with_label ("Teacher");
gtk_box_pack_start(GTK_BOX(box),check,TRUE,TRUE,0);
check = gtk_check_button_new_with_label ("Student");
gtk_box_pack_start (GTK_BOX(box),check,TRUE,TRUE,0);
radio = gtk_radio_button_new_with_label (NULL,"Chinese");
gtk_box_pack_start (GTK_BOX(box),radio,TRUE,TRUE,0);
group = gtk_radio_button_group (GTK_RADIO_BUTTON(radio));
radio = gtk_radio_button_new_with_label (group,"Math");
gtk_box_pack_start (GTK_BOX(box),radio,TRUE,TRUE,0);
group = gtk_radio_button_group (GTK_RADIO_BUTTON(radio));
radio = gtk_radio_button_new_with_label (group,"English");
gtk_box_pack_start (GTK_BOX(box),radio,TRUE,TRUE,0);
gtk_widget_show_all (window);
gtk_main();
return 0;
}
```

5. 创建一个带菜单和快捷工具栏的窗体。

请参考例 12.21 和 12.23。